Alexander Kainz, Bsc

SteamVis: A Tool for Collecting and Analyzing Data of the Game Distribution Platform Steam

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Johanna Pirker, BSc

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl-Ing. Dr. Stefanie Lindstaedt

Graz, February 2022

Affidavit

Abstract

Over the past decades, a rapid growth in the video game industry happened. The success of video games can partly be traced back to game distribution platforms like Steam, which allow buying and downloading games. As a result, a rising interest in such platforms can be detected among researchers, business analysts, and game developers. Steam distributes over 60,000 games and is well suited for studying user behavior as it provides access to user-written game reviews. Furthermore, game data and reviews can be collected from Steam via diverse APIs or directly from the Steam website. Steam data has already been exploited by many researchers to achieve various goals. What the conducted studies have in common is that they require a methodology to collect, store, and analyze the data. Although the methodologies of many studies are similar, there is no common approach.

Thus, this thesis introduces SteamVis, a tool that eases the procedure for collecting and analyzing data from Steam. The tool can collect data associated with a set of games fulfilling specific characteristics. For each game in the set, the tool extracts game-related information from the Steam website. Furthermore, the tool has a feature for collecting user-written game reviews with additional metadata (author, review helpfulness, review votes, etc.). Moreover, the tool handles the storing of the data and has support for performing analysis tasks on the data, where data can be visualized via charts or tables.

To obtain feedback on the tool, a first study with six domain experts was carried out. In the study, the participants had to test the tool by performing several tasks. The main goal was to detect inconsistencies, misconceptions, software bugs, and missing features. Furthermore, standardized questionnaires were used to assess the tool's usability and the users' emotions while using the tool. The results indicate good to excellent usability but also reveal improvements that can be applied to enhance the tool in the future.

Acknowledgments

First, I want to thank all people who supported me during my time at the Graz University of Technology or played an essential role in creating this work. A big thank you goes to my supervisor Johanna Pirker, who gave the work a direction, helped me structure this work, provided me with ideas, and was always open for questions, even during non-work hours.

Also, I appreciate the time fellow students and researchers invested by testing and giving feedback on my proposed tool for collecting and analyzing data from the game distribution platform Steam.

Furthermore, I want to thank my parents and girlfriend, who helped me through this work-intensive time. Without them, it would not have been possible for me to study Computer Science from a financial and mental aspect. They told me to never give up and backed me up during hard phases in my study.

At last, I want to thank my long friend Johannes Kopf and my girlfriend Petra Rainer for proofreading this work.

Contents

List of Figures

List of Tables

Listings

1. Introduction

The video game market belongs to the fastest-growing industries. In the year 2020, the market had a value of approximately 159.89 billion U.S. dollars, and it is estimated to grow up to a value of 268 billion U.S. dollars until the year 2025 (Juniper Research, 2021). Steam[1], a service from Valve Corporation[2], is one of the most prominent game distribution platforms. It provides access to more than 60,000 games, where the number of annual releases increased over the past few years (Steam Spy, 2021). The growing nature of Steam can also be observed by the increasing number of peak concurrent users over the years and months. Whereas a record taken in January 2013 shows a peak concurrent number of users of about 6.61 million, a record taken in September 2021 already shows a peak concurrent number of users of about 26.09 million users (Steam, 2021).

The high number of games and users leads to the generation of a large amount of data. This data can be accessed via diverse APIs or the Steam website itself. These sources allow researchers, business analysts, and game developers to analyze and explore data in many different ways or conduct experiments. As Steam provides user-written reviews, its data is especially useful for studying user behavior.

A lot of studies have already been performed that utilize data from Steam. The studies concern different topics like review sentiment (Bais et al., 2017; Ji, 2019), recommendation systems (Kamal et al., 2020; Wang et al., 2020), review helpfulness (Barbosa et al., 2016; Eberhard et al., 2018; Kang et al., 2017), and so on. Although the approaches and methodologies for analyzing the data differ, the procedures and sources for collecting data from Steam

[1]https://store.steampowered.com/
[2]https://www.valvesoftware.com/de/

overlap. Thus a common tool that eases collecting and analyzing data from Steam becomes valuable.

1.1. Objectives

Thus, the goal of this work is to provide game researchers with a tool that allows them to collect and explore data from the game distribution platform Steam. The aim is to save the researchers time and effort by implementing commonly required procedures in advance.

The tool should enable users to collect general game data like the title, publisher, developers, associated genres, user-defined tags, and other game-related information. Moreover, it should be possible to collect reviews of games. Filters should allow restricting the data collection to a specific set of games. Similarly, the tool should also have an option to collect only reviews that fulfill selected criteria concerning the language and creation time.

To enable researchers to get quickly started with a project, the tool should have analytic capabilities. On the one hand, the tool should give a quick overview of the games with minimal required user interactions. On the other hand, the tool should also provide advanced features allowing the definition of custom analysis tasks. Moreover, it should be possible to visualize data via charts or tables. In addition, storing the charts to images or the table content in a CSV file should be feasible.

1.2. Methodology and Structure

The thesis is split into five major parts. The first part is about the background and related work. The second part describes the design of the tool. The third part focuses on the implementation. The fourth part discusses an example use case utilizing the tool. The fifth evaluates the results of a user study that is conducted to obtain feedback on the tool. The last part discusses learned lessons, outlines future work, and rounds up the thesis via a conclusion. The workflow is visualized in Figure 1.1.

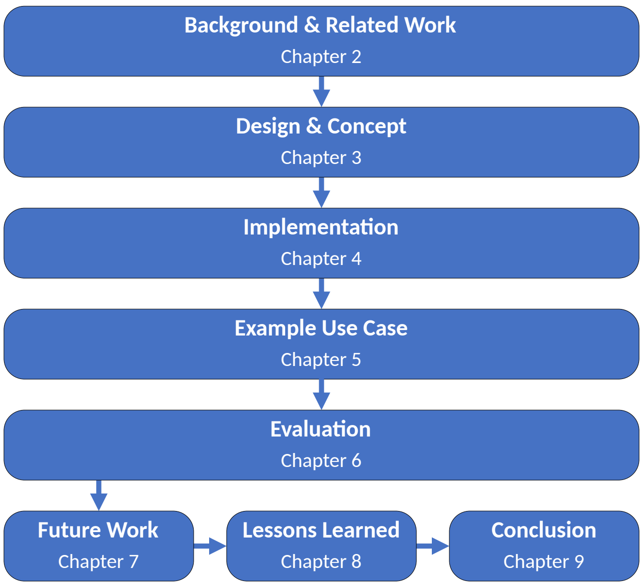


Figure 1.1.: Structure of the thesis.

Chapter 2 provides insights into existing game review platforms and compares them against each other. Furthermore, it explores studies that collect and analyze data from the platform Steam or Metacritic. One section focuses on the methodologies for collecting data from Steam by describing and listing data sources that were utilized in different studies. Another section gives an overview of existing analytic tools and their features.

Chapter 3 can be seen as the starting point of the implementation and defines the objective, user target group, and requirements of the tool. Moreover, important decisions are made throughout the chapter, including the programming language, data storage, and user interface framework. To get a high-level overview of the tool, the chapter further provides a logical architecture with a focus on data collection, data analysis, and data management. Build on top of the architecture, a potential design of the tool is given. The last section describes two design patterns that are used in the implementation process.

Chapter 4 details the different implementation steps. It starts by describing how the user interface is created and how it is given its style. Afterward, the chapter describes the data structure that is used to store the collected data from Steam by introducing an SQL schema. Furthermore, it explains the building process of the SQLite Python library to support mathematical functions and to have text-based search capabilities. Having the user interface and data storage described, the chapter presents the architecture for collecting and analyzing data. Then, the data collection and data analyzing implementations are explained in two separate sections. Furthermore, the chapter gives a short introduction on how the error handling is done and the last part describes one possibility to distribute the tool to the end-user.

Chapter 5 provides an example use case of the implemented tool. It describes an ongoing study, where the data collection, as well as the analysis, is directly done in the tool.

Chapter 6 evaluates the results of a user study that was conducted to obtain feedback on the tool. It describes the evaluation methodology, procedure, material, and used questionnaires. The results are analyzed concerning usability and emotions. Furthermore, the feedback and suggested improvements of the participants are summarized.

Chapter 7 provides an overview of lessons that were learned during the creation of this work concerning theory, development, and evaluation. It has a look at different challenges and problems that occurred and how they were solved. Chapter 8 gives an future outlook taking the evaluation results into account. It lists technical improvements and features that should be part of a later version of the tool. Finally, Chapter 9 summarizes the content of the work by giving a conclusion.

2. Background & Related Work

Game review platforms provide a large amount of information that can be explored in many different ways. However, the game review platforms differ in their characteristics. While some only provide reviews written by professional critics, others list user-generated ones. In addition, professional reviews might be biased as they might not properly capture the opinion of the majority of the players. Another reason why user-generated content is important is that it enables developers and publishers to understand and connect with their community. Thus, user reviews enrich market research and are a major factor in the decision-making process of game development studios. As a result, this work mainly focuses on platforms that provide user-generated reviews rather than content created by professional critics. Still, it is crucial to understand the main differences between different game review platforms. Therefore, this chapter starts by giving an overview of the game review platforms and their characteristics in the next section.

2.1. Game Review Platforms

Different web entries (Leclair, 2021; Stegner, 2021) provide a list and a ranking of the most popular game review platforms. To determine the key differences between the platforms or websites, a manual investigation was performed. The most important characteristics of these platforms include the type of reviews, the ranking system, the information they provide, and which games they review.

Regarding reviews, we can differ between professional and unprofessional ones. Professional reviews are written by critics and are published on sites like GamesRadar[1], Gamespot[2], or Destructoid[3]. In contrast, unprofessional reviews reflect the opinion of general users and are usually short. Steam[4] and Oculus Store[5] are two game distribution platforms that allow writing unprofessional reviews. While in Steam, it is only possible to mark reviews as positive or negative, Oculus Store uses a 5-stars rating system. In the case of the Steam platform, the overall score of a game is calculated as the ratio between the positive and total number of reviews, and in the case of the Oculus Store, the individual 5-star ratings are averaged. Other platforms like OpenCritic[6] and Metacritic[7] aggregate the ratings from various sources and calculate an overall professional score, although the way on how the overall score is determined differs.

Another difference between game review platforms is that some distribute games (e.g., Steam, Oculus Store). However, such platforms in general only allow to rate the games they offer in their store. Furthermore, the platforms (e.g., Metacritic) are often not limited to games but provide reviews and other content about TV shows, movies, or music. However, reviews are not limited to text only. They can also be provided by speech (e.g., podcasts) or videos (e.g., YouTube).

As this work focuses on platforms that provide user-written reviews, platforms that only publish reviews from professional critics are neglected for further discussion. Both Metacritic and Steam allow writing non-professional reviews for video games. A self-conducted literature research on Google Scholar revealed that their data has been collected and assessed by game researchers in various ways and for different purposes. To get an impression of what has been done in the past, and to get a feeling of how game and review data can be explored, the next two sections provide an overview of studies that exploited data from Metacritic or Steam.

[1]https://www.gamesradar.com/
[2]https://www.gamespot.com/
[3]https://www.destructoid.com/
[4]https://store.steampowered.com/
[5]https://www.oculus.com/experiences/quest/
[6]https://opencritic.com/
[7]https://www.metacritic.com/

2.2. Metacritic Analysis

Metacritic is an online platform that aggregates and summarizes reviews for different categories including games, movies, TV shows, and music. It shows the Metascore and the user score for each product, where the former one is based on reviews from professional reviewers, which are collected from various sources (e.g., websites). The rating score (e.g., 0-5 stars, 0-10 rating) of each source is converted to a common 0-100 score. The Metascore is created by taking the weighted average of the converted scores, where the weight depends on the quality of the source. In contrast, the user score is calculated from the ratings of unprofessional reviewers. Hereby, users can provide a textual description along with their rating (0-10 score). The user score is independent of the review text and user characteristics, meaning that each review has the same influence on the overall rating. Depending on the score, a review is marked as negative (0-4 score), mixed (5-7 score), or positive (8-10 score) indicated with the colors red, orange, and green. Like Steam, Metacritic provides meta information. For the games category, this includes the developer, release date, supported devices, a summary, the associated genres, the number of ratings the user score is based on, the number of critical reviews the Metascore is based on, and so on. Like on Steam, users can mark reviews of authors as helpful or unhelpful and sort them based on the helpfulness (Kasper et al., 2019; Strååt et al., 2017). In this chapter, studies that use game-related data from Metacritic are discussed. It is structured into different topics concerning review helpfulness, review sentiment, Metascore vs. Steam user score, and the influence of reviews on video game success.

2.2.1. Review Helpfulness

Kasper et al. (2019) analyze the impact of the genre, score, and review content on the number of helpfulness votes and the helpfulness prediction. In their study, they extract a set of features regarding writing style (e.g., character count, sentence count, line break count, readability metrics, etc.), sentiment (compound score, positive/negative/neutral scores calculated by using VADER[8]), and content (censored word count, integer count, percentage symbol count, etc.). By performing a correlation analysis, the authors detect that there is a strong correlation between the review score and the ratio between the number of helpfulness up- and down-votes for games with a specific genre. In addition, the authors perform a prediction experiment and compare the performance of models neglecting different features. Using a model with only text-based features performs worse compared to a model using only the score as a feature. The best performance is achieved by a model that uses both text-based features as well as the score. For feature weighting purposes and to visualize the importance of different features, the authors use a Gradient Boosting Classifier. The resulting averaged weights over the genres, using a feature space consisting of the text-based features and the score feature, indicate that the review score has a high influence on the helpfulness ratio. Therefore, it makes sense to include it as a feature for predicting the helpfulness of user reviews.

While Kasper et al. (2019) only uses review sentiment as an input feature, others use it as the main tool for finding data correlations.

2.2.2. Review Sentiment

The study of Strååt et al. (2017) investigates if there exists a direct correlation between the rating and specific aspects of the user reviews. Explicit aspects are described as words that occur frequently over reviews and are relevant to the product (e.g., gameplay). These words are extracted from the reviews by performing a frequency analysis and neglecting non-domain relevant words. In addition, phrases that have a strong connection to an extracted word but do not use the word in the context, are defined as intrinsic aspects. The whole process is known as aspect-based sentiment analysis (Pontiki et al., 2016). The authors use it on the Metacritic reviews of the PC versions of the first three games of the Dragon Age franchise, namely Dragon Age Origins, Dragon Age 2, and Dragon Age Inquisition. They focus on the three explicit aspects with the highest frequency (Story, Combat, and Character). Reviews that are not connected to one of these three aspects are neglected

[8]https://github.com/cjhutto/vaderSentiment

for further investigation. The remaining reviews are then submitted to a crowd-sourcing online platform, where inspectors determine for each review if the associated aspects have been used in a bad, neutral or negative way. The numbers are then put into relation with the Metacritic ratings (negative, mixed, positive) on a per aspect and game basis. In all cases, a connection of the aspects with the rating can be observed. However, rather than finding a correlation between the rating and review aspects, other studies try to detect if there is a mismatch between the review rating (user score) and Metacritic's professional score.

2.2.3. Metascore vs Steam User Score

Park and Byun (2016) investigate if there is a difference between Steam's user score and the Metascore (professional score) for games published on the platform Steam. At the time the study was conducted, twelve different genres existed. However, the genres "Free to Play" and "Early Access" were neglected as part of the study. For each game, the authors stored the associated genres. Other information, including the name, year, Metascore, and user score, was gathered from Steam. To make the user score (0-10 score) comparable with the Metascore (0-100 score), the user score was multiplied by ten. The results from correlation analysis between the user score and Metascore per genre show that in the case of the genres "Indie" and "Casual", there is no significant difference between the two scores. In addition, the authors detected that the Metascore tends to be higher than the user score. If and to what extent the two score types influence the success of a video game is discussed next.

2.2.4. Influence of Reviews on Video Game Success

The research of Sherrick and Schmierbach (2016) examines if the Metascore, i.e., the score aggregated from professional reviews, and the user score have an impact on the unit sales of video games and how the influence changes over time. Additionally, the authors investigate the relationship between the amount of advertising for a product and the unit sales. The professional score (Metascore), as well as the user score and their corresponding number of reviews, are gathered from Metacritic. For obtaining the sales data, VGChartz[9] providing the sales data of the first ten weeks after a product is released and on an annual basis, is used. The advertising budget for each game is gathered from Ad$Spender[10], which the authors utilize to calculate the overall advertising expenses per video game. The resulting coefficients of a regression analysis indicate that the Metascore has a consistent influence on unit sales. Furthermore, the results show that the number of reviews the Metascore aggregates has an high influence on the sales in the first few weeks but decreases over time. However, the user score only shows a weak relationship with the number of sales. The advertising expenses have a significant influence on the sales in the weeks two to ten, but otherwise, the influence seems to be small. The reason for this might be that most advertising is done before and immediately after the product release.

This section was about studies that collect and use data from Metacritic. Similarly, the next section is about studies that collect and exploit data from the Steam platform.

2.3. Steam Analysis

Steam is the biggest video game distribution platform. It was developed by Valve Corporation and was released in the year 2003. At the time of 1st October 2021, it allowed downloading over 50,000 games (Clement, 2021). For each game, Steam provides game-specific information. This includes the title, names of the developer and publisher, release date, price, rating, associated genres, user-defined tags, and a detailed textual description of the game itself. Furthermore, Steam allows the users to write reviews and rate the games as positive or negative. In addition, users can mark a review as helpful or unhelpful, which creates a metric that eases the decision to read a specific review or not. The review data and general game data provide researchers with a wide range of possibilities to study the characteristics of games and also the player behavior in various manners and by using

[9]https://www.vgchartz.com/
[10]https://johnson.library.cornell.edu/database/ad-spender/

different approaches. This section provides an overview of studies that have been performed using the information provided by the Steam platform and gives insights about different approaches to analyze the game and review data. Like in the previous section, we start by investigating studies that aim to analyze the helpfulness of user reviews.

2.3.1. Review Helpfulness

The study of Eberhard et al. (2018) does a general investigation of the helpfulness by trying to detect differences between the helpful and unhelpful reviews. The authors perform a binary prediction experiment. This is done by extracting different features from the reviews regarding structure, formatting, readability, sentiment, and content. Furthermore, features from additional information, like Early Access state, playtime, products owned by the reviewer, are extracted. Next, the authors assess the review helpfulness by first separating their review dataset into the three sets Unhelpful, Helpful, and Top, based on the number of votes on the reviews. For evaluating if there are significant differences between the individual features, the authors use the Mann-Whitney U test. To predict if a review is helpful or unhelpful, a pairwise classification of the three sets is performed, where a random forest with 20 decision trees is created. By investigating the feature differences, the authors have detected that the review length, as well as the time spent in a game, have a strong impact on the helpfulness. Concerning the prediction experiment, a prediction score higher than the random baseline (0.5) has been achieved for all three experiments (Unhelpful vs. Helpful, Unhelpful vs. Top, Helpful vs. Top). In the case of Unhelpful vs. Helpful, a preference towards the negative class has been detected, where the authors see the atypical nature of the review texts (e.g., only hyper-reference to an external source as review text) of the Unhelpful class as the reason for it. The key findings of this study indicate that high-rated reviews tend to be longer, more complex, more critical, and more detailed.

A similar study, but using another approach, was performed by Barbosa et al. (2016). Hereby, an analysis on a Steam review dataset is conducted using a Multilayer Perception Artificial Neural Network (MLP ANN). In the study, the MLP ANN is designed to consist of an input layer, output layer, and only one hidden layer. For the input, features are extracted from the review dataset. This includes features about the author of the review like the number of friends and reputation, where a high reputation indicates that the author has already written high-rated reviews in the past. Furthermore, features about the textual content and semantic, are extracted. In addition, also review metadata (average rating of the game, review creation date) is considered as input for the MLP. Performing a feature selection analysis revealed that the word count, the number of the author's friends, and the author's reputation are important factors that influence the helpfulness of a user review. That a higher word count is connected to helpful reviews is also consistent with the findings of Eberhard et al. (2018).

Kang et al. (2017) compares two different data mining approaches for predicting the helpfulness of Steam reviews. As in the two previously mentioned studies, features are extracted from the review dataset and are then used for creating the models. The first approach called CART (Lewis, 2000) is a decision tree-based algorithm, where the dataset is divided into smaller subsets. The algorithm tries to arrange the features that provide the best separation of the dataset. The tree grows until the child nodes have the same category as their parent node. The second approach uses an MLP ANN with one layer of four hidden neurons. The accuracy of the two models was compared using the Sum of Squared Error (SSE) and the Mean Absolute Error (MAE). By testing the models with the test set (10%), the CART approach achieved higher accuracy than the MLP ANN approach. The CART approach shows that the number of total votes, followed by the number of the author's friends has the highest influence on the review helpfulness.

In contrast, other studies do not aim to find out if a review is helpful or unhelpful but attempt to detect if a review is positive, negative, or neutral. In other words, they aim to extract the sentiment of a review.

2.3.2. Review Sentiment

Ji (2019) uses natural language processing (NLP) to detect if Steam reviews are positive or negative. To achieve this task, the authors make use of two different approaches, namely naive Bayes and Support Vector Machines (SVMs). In the data pre-processing step, stop words are removed from the review comments, and the remaining words are reduced to their word stem, also known as stemming. To assess the performance of their implementation, the authors calculate the prediction accuracy for each selected game and compare them to the prediction accuracy of Python's scikit-learn library[11] implementing the same approaches. The result shows that the authors' implementation achieves similar accuracy as the scikit-learn implementation if a high percentage of the reviews are positive. In contrast, if the ratio between positive and negative reviews goes towards one (# positive reviews / # negative reviews = 1), the performance degrades.

A similar study by Bais et al. (2017) compares the performance of five different approaches with each other. These approaches are based on different methods including naive Bayes, lexicon score aggregation, modified Turney's algorithm, logistic regression, and linear SVMs. In contrast to the study of Ji (2019), the study also takes other meta-features like the review helpfulness, review funny votes, and the number of hours played by the review creator into account. In addition, term frequency-inverse document frequency (TF-IDF) weighting is applied to the individual stemmed words. The best performance is achieved by the implementation based on the SVM method.

Up to this point of the section, all the studies use reviews in one way or the other. However, other studies do not use reviews at all but focus on game features. Especially genres and tags are features that have a descriptive power of games on Steam.

2.3.3. Genres & Tags

While genres are defined by Steam or the publisher itself and are fixed, tags are user-generated. As mentioned by Windleharth et al. (2016), tags or genres are used for describing media to other people or to help them to find the content of their interest in a fast manner. In their study, they performed a conceptual analysis of the user-generated tags on Steam. In

[11]https://scikit-learn.org/stable/

total, 294 different tags could be detected in the dataset collected on March 11, 2015. Having these tags, the authors sorted them into categories based on observations obtained from frequently searching for games with a specific tag during the sorting process. In the end, they came up with 29 different categories. Among others, this includes the categories "Gameplay Genre", "Mechanics and Input", "Progression", "Narrative Genre", "Theme, Setting and Mood", "Visual Style", "Tropes", "Media Style" and so on. Hereby, the authors provide a description of the individual categories and also compare their set of categories with the game entities of the Video Game Metadata Schema (VGMS) created by the University of Washington Information School Game Research Group and Seattle Interactive Media Museum (Group & Museum, 2019). A high portion of the authors' set overlaps with the game entities of the VGMS. However, based on their analysis, they suggest adding three additional entities concerning mechanics, user interaction, and evaluation.

Regarding genres, one that got more attention over the last years is named "Early Access". On Steam, this genre is assigned to games that are published in the so-called Early Access state, where unfinished games can be bought and played by users. Such a system helps developers to get feedback in an early state by directly communicating with their customers. On the other hand, the customers can influence the direction of further game development in their favor. The empirical study conducted by Lin et al. (2018) aims to assess characteristics of Early Access games published on Steam. The results show that a lower percentage of players write reviews in the Early Access state than directly after the game leaves the state. However, the authors note that reviews tend to be more positive in the Early Access state. This indicates that the customers show more acceptance if they know that the product is unfinished and the goal of the developers is to get feedback to improve the experience. Another interesting finding of the study is that the Early Access model is often used by solo developers or small game studios.

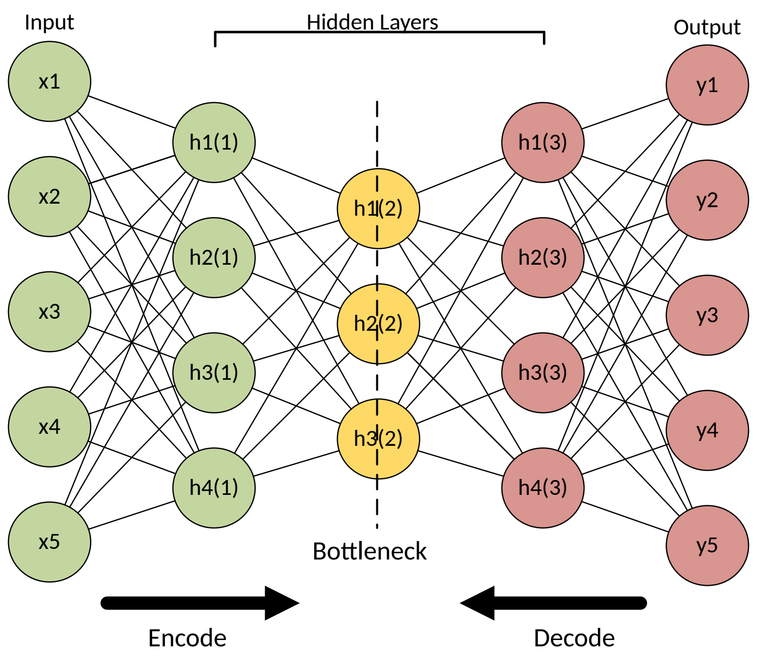


Figure 2.1.: Visualization of an autoencoder with three hidden layers. The layer in the middle corresponds to the bottleneck.

2.3.4. Recommendation Systems

Recommendation systems are used to recommend products to customers based on their preferences and their already performed activities. In the case of streaming platforms like Amazon Prime, the view history corresponds to the already performed activities. Such systems should help users to find the content of interest fast. Furthermore, video game distribution platforms like Steam use recommendation systems to make the decision, which games to buy, easier for the players.

Wang et al. (2020) proposes a recommendation system called STEAMer, which makes use of the user data in combination with a deep autoencoder. A deep autoencoder is a type of artificial neural network with the aim to find key features of the input data. It consists of several hidden layers, where the number of neurons in the hidden layer is less than the number in the input layer. The autoencoder can be subdivided into three parts, namely the encoder, bottleneck, and decoder. The encoder compresses the input data, the bottleneck corresponds to the hidden layer with the least neurons and the decoder decompresses the encoded data again. A visualization of a deep autoencoder with three hidden layers can be seen in Figure 2.1. For training, the deep autoencoder, Wang et al. (2020), used a set of game and user-related features including genre, rating, developer id, publisher id, playtime, and friends. The performance of the model was compared with two other models based on deep neural networks and achieved the best results. In the conclusion, the authors mention that additional user data as features improves performance.

In another study by Kamal et al. (2020), three recommendation systems are implemented and compared against each other. The first system uses the genres assigned to the individual games and the user's preferences extracted from the playing history. The second system extracts the topic from the game descriptions instead of directly using the genres. And the third one is a hybrid system of the genre and topic-based approaches. For predicting the rating of a game, the k-nearest neighbors (KNN) algorithm is used. Their evaluation shows that the genre-based approach performs better compared to the topic modeling and the hybrid approach. However, the authors note that their implementation performs worse compared to implementations of previous studies using different approaches and parameters. They conclude that the genre/topic is not suitable for building a game recommendation system.

However, when building a recommendation system, the geographic location of a user might be of importance as well, as they might tend to prefer specific game genres over others. Taking geographic features into account might enhance the genre-based recommendation system of Kamal et al. (2020). In addition, recommendation systems might be biased towards a specific country with more players. As a result, the next part looks at the user distribution and genre popularity in a geographic manner.

2.3.5. Popularity

The study of Toy et al. (2018) shows the geographical distribution of users, genres, and genre popularity. For this, the authors make use of heat maps to visualize the distribution. Regarding the number of users and the number of downloaded games per country, the United States stands out. However, the genre popularity per country, determined by the number of game downloads per genre and country, indicates that there are only minor trend differences. It must be noted that this study is based on an outdated Steam dataset created in the year 2016 by O'Neill et al. (2016a). A relative distribution of Steam users worldwide from the year 2018 (Clement, 2018) already differs from the results. While the United States is still the most dominant country in terms of the number of users, a big rise in the popularity of Steam can be detected in China.

While Toy et al. (2018) looks at the genre popularity of different countries, the authors do not assess which factors actually make a game popular and worth playing. However, studies exist which aim to create a parameter that is representative for the quality of a video game.

2.3.6. Playability

User reviews on games can be exploited in many ways to assess corresponding game and user characteristics. An important video game characteristic is playability which has a deep connection to the quality of a video game in terms of usability and user experience (Sánchez et al., 2012). Li et al. (2021) proposes an approach to analyze the playability by extracting user opinions from a large set of game reviews from platforms like Steam and by further analyzing them via their suggested evaluation framework. The simplified idea behind the framework is to classify the reviews into a set of playability perspectives (e.g., gameplay, functionality, usability (Paavilainen, 2020)), generating an overall opinion per category and reporting the results. In addition, the framework performs a topic modeling on the classified reviews to extract the advantages, disadvantages, and topics for each playability perspective, and adds the results to the playability report.

So far, only studies have been investigated that do not make a difference between PC games and Virtual Reality (VR) games. As the Virtual Reality market is growing, which can for example be seen in the increasing number of unit sales of VR headset devices during the Covid-19 pandemic (GfK, 2020) or the rising gaming revenue in the last few years (PwC, 2020), the exploration of VR games and applications becomes more important. Thus, studies, which utilize data from various platforms and purely focus on analyzing VR games, are independently discussed in the following section.

2.4. VR Games Analysis

There is especially a lack of studies that investigate the impact of VR on the users' health by using game or review data from platforms like Steam or Oculus Store. In addition, the influence of the coronavirus on Virtual Reality in terms of user behavior seems rather unexplored. Still, a few studies exist with the goal to explore VR with respect to education (Radianti et al., 2021; Smutny et al., 2019), mental well-being (Fagernäs et al., 2021), and market research (Ho & Zhang, 2020).

Other studies, like the study of Foxman et al. (2020), try to get more insights into the genre distribution of VR games regarding downloads and user ratings by using game data from Steam. In addition, the authors compare their findings with non-VR applications. The results indicate that the genres Simulation, Action, and Shooter are the genres with the most downloads, while the genres Action and Music are the highest rated ones. Compared to non-VR games, VR games tend to be less popular for games with a longer playtime (e.g., Shooter). The authors mention cybersickness, eyestrain, and heat of a VR headset as possible reasons for this. Furthermore, the study shows that non-VR games are in general rated higher compared to VR games.

Epp et al. (2021) investigates the characteristics of VR games on Steam with respect to price, Head-mounted display support, motion tracking support, play area support, and the number of updates. In addition, they analyze the player reviews with the aim to discover the complaints in connection with

VR games. Their results show that the complaints comprise the lack of game content, the high prices, cybersickness (LaViola Jr, 2000), lack of community, game-specific complaints, controls, and optimization. However, the authors note that VR-specific complaints (cybersickness, controls, optimization) seem to be less important to the players than the ones that also occur for non-VR games.

Virtual Reality is deemed as a promising technology for education (Radianti et al., 2020). The educational VR literature review of Hamilton et al. (2021) shows that studies exist for different subject areas. However, it seems that most performed studies apply a form of user evaluation method (e.g., pre-posttest design, multiple-choice questionnaires). By performing a search for related studies on Google Scholar that utilize data from a game review platform or game distribution platform, only a small number of scientific papers could be detected. The studies of Radianti et al. (2021) and Smutny et al. (2019) aim to determine the application/subject domain of educational VR games. While Radianti et al. (2021) use data from Steam, Vive[12] and Google Play[13] as input for their study, Smutny et al. (2019) only use data from the Oculus Store. Although the names of the subjects differ in both studies, educational VR applications related to space and nature seem to be popular among developers.

2.5. Steam Data Retrieval Methodologies

What most of the previously discussed studies that are related to Steam have in common is that they need data about the games, reviews, or users. Depending on the different needs, the authors choose different methods to retrieve the data directly from Steam or a third party. In addition, different approaches for storing the data are used, although in most studies the storing technology is not mentioned. Table 2.1 shows an overview of which data sources and storage technologies are used by different studies. Two x-marks in the case of the storage indicate that the storage technology is

[12]https://www.viveport.com/
[13]https://play.google.com/store

Paper	**Source**					**Storage**	
	Steam Website	**Steam API**	**Steam Spy**	**Steam DB**	**Database O'Neill**	**JSON**	**RDBMS**
Toy et al., 2018	✗	✗	✗	✗	✓	✗	✓
Wang et al., 2020	✗	✗	✗	✗	✓	✗	✗
Kamal et al., 2020	✗	✗	✗	✗	✓	✗	✗
Lin et al., 2018	✓	✗	✓	✓	✗	✗	✗
Windleharth et al., 2016	✓	✗	✗	✗	✗	✗	✗
Sobkowicza and Stokowiec, 2016	✓	✗	✗	✗	✗	✓	✗
Ji, 2019	✓	✓	✗	✗	✗	✓	✗
Bais et al., 2017	✓	✗	✗	✗	✗	✓	✗
Eberhard et al., 2018	✓	✗	✗	✗	✗	✗	✗
Kang et al., 2017	✗	✗	✗	✗	✗	✓	✗
Barbosa et al., 2016	✓	✗	✗	✗	✗	✗	✗
Lin et al., 2019	✓	✗	✓	✓	✗	✗	✗
Sifa et al., 2014	✗	✓	✗	✗	✗	✗	✗
Sifa et al., 2015	✗	✓	✗	✗	✗	✗	✗
Epp et al., 2021	✓	✗	✓	✓	✗	✗	✗
Radianti et al., 2021	✓	✗	✗	✗	✗	✗	✗

Table 2.1.: Data sources and storage technologies of Steam related studies. Two x-marks in the case of the storage indicate that the storage technology is unknown.

unknown. The following part describes the five main data sources in more detail and what information they provide.

2.5.1. Steam Data Sources

By investigating studies that require current or historic data from Steam, five main data sources could be detected, which differ in the data they provide. This data sources are the Steam Website, Steam API[14], Steam Spy[15], Steam DB[16] and the database of O'Neill[17].

Steam Website

A high portion of the investigated studies in Table 2.1 creates a custom web crawler to gather information directly from the Steam website. For the purpose of data retrieval, Python seems to be the favored programming language. Hereby, the authors make use of different Python libraries (e.g., Sobkowicza and Stokowiec (2016) - Selenium[18], Radianti et al. (2021) - BeautifulSoup[19], Barbosa et al. (2016) - Scrapy[20]) for accessing the web pages and extracting information by parsing the HTML content. Although accessing the information on Steam by writing custom web crawlers is more complex compared to the Steam API, the big advantage is that the content is not limited to the information the API provides and it is possible to crawl further data which might not be provided by the API.

[14]https://partner.steamgames.com/doc/webapi_overview?
[15]https://steamspy.com/
[16]https://steamdb.info/
[17]https://steam.internet.byu.edu/
[18]https://pypi.org/project/selenium/
[19]https://www.crummy.com/software/BeautifulSoup/
[20]https://scrapy.org/

Steam API

The Steam API provides interfaces for accessing different features via HTTP requests. While some Web API methods are public, others are private and require an additional key. This is especially done to limit access to sensitive data (Valve, 2021a). For instance, the API provides a method for obtaining the IDs and names of all currently available games (Valve, 2021c) and a method to get user reviews (Valve, 2021d) for a specific game ID. Hereby, the HTTP responses are encoded in three different formats, namely JSON, XML, or VDF depending on the *format* parameter of the HTTP request. The default encoding is JSON and is used if the *format* parameter is not provided or is set to *json* (Valve, 2021b).

Steam Spy

Steam Spy is a third-party project that continuously extracts data from Steam. It provides general game information, including the associated genres and tags, supported languages, current price, developer, publisher, release date, playtime statistics (average, median), and other game-related information. Furthermore, it monitors the tags over time and provides information about the owners. While viewing general game information is unrestricted, data about the owners, tags over time, and deeper insights into the playtime, is restricted. For unlocking data, there are three price categories (Basic, Indie, and Pro). Depending on the subscription, different features are unlocked. According to the creator of Steam Spy, Sergey Galyonkin, individual application data is updated on a three-day basis, which means that it is unreliable for games on Steam that have just been released (Galyonkin, 2021b). Furthermore, as it is not possible to gather data from all user profiles, some data is based on an estimation of a random set of users (Galyonkin, 2021b; Lin et al., 2018). To get general information about games, Steam Spy provides a simple API, where the data can be accessed via HTTP requests (Galyonkin, 2021a).

Steam DB

Steam DB is a third-party website that provides deeper insights by constantly crawling data from the Steam platform. It lists the current number of players playing a game, and in addition, provides historic time charts showing the concurrent number of players over time. Furthermore, it aggregates game-specific information from Steam Spy (e.g., playtime estimation) and Twitch (e.g., # of current viewers for a specific game channel). Like Steam Spy, Steam DB provides general game information (release date, genre, supported languages, Metascore, etc.). In addition, it tracks the game price per concurrency on Steam over time, which data is, for example, used by Epp et al. (2021) and Lin et al. (2019), Lin et al. (2018).

O'Neill Database

The database of O'Neill (O'Neill et al., 2016b) is a 17GB SQL dump file, which can be downloaded for free. The information the database contains was mainly retrieved using the Steam (Web) API. It has a table about the general game information (game id, title, game type, price, release date, rating, required age, multiplayer indication). Furthermore, it provides tables about developers, publishers, achievements, and genres, where each table has a field *appid* indicating the corresponding game. Concerning user information, it provides a table about friend relationships. Each user is identified via a unique Steam ID. In addition, the database has a table containing information on how much time a user spent playing a specific game. The database was created as a part of the study by O'Neill et al. (2016a), where the authors collected information of 108.7 million user accounts and data about their owned games, playtime, and friend relationships. As the process of collecting public user data, game ownership information, and friend relationships requires a large amount of API calls and the number of API requests is limited by the Steam Web API terms of use to 100,000 calls per day, the dataset is used in studies where such information is required (e.g., Kang et al. (2017) and Wang et al. (2020)).

2.5.2. Storing Technology

Studies (Bais et al., 2017; Ji, 2019; Kang et al., 2017; Sobkowicza & Stokowiec, 2016) that mention how the retrieved Steam data is stored for further processing, use text files with Javascript Object Notation (JSON) formatting. However, studies like the one from Toy et al. (2018) use a Relational Database Management System (RDBMS). One of the main reasons why the authors use JSON encoded files for storing their data might be that the default response encoding of an API like the Steam API is JSON encoded, too. Another reason might be that storing data in JSON files is easier than setting up a relational database. However, for connecting content and querying specific data, relational databases are preferable and faster as they usually use advanced indexing. Furthermore, in the case of simple JSON files, the whole file is read, although only a small amount of data might be needed. However, as performance is no issue in most of the mentioned studies in Section 2.3 and Section 2.4 and relationships between the collected data are comprehensible, JSON might be the right choice.

Although studies partly mention how the data is stored and which programming language is used for the purpose of collecting and analyzing data, they do not implement a tool with a GUI on their own. However, as the goal of this work is to create a user interface that should ease collecting and analyzing data from Steam, the next part introduces a few existing and prominent analytic tools.

2.6. Analytic Tools

For analyzing and visualizing data, diverse analytic tools exist that differ in the provided features, price, and supported data inputs. Tableau[21] is considered to be the world's leading analytic platform. It is easy to use, has a large community, and has good support. Furthermore, it can connect to a wide range of data sources, including Excel files and relational databases. Their visualizations are highly regarded for their customizability, interactability, and appealing design. Mentioned disadvantages of Tableau concern

[21]https://www.tableau.com/

high and inflexible prices, required knowledge for using complex features, and slow speed when working with large datasets. However, Tableau also has a free version with limited functionalities for non-profit and academic purposes (Kaur, 2017; Scheiner, 2021; Scott, 2017).

An alternative tool is PowerBi[22], which in contrast to Tableau supports natural language queries. Concerning other features the two platforms are similar. However, Tableau is considered to handle large datasets more efficiently. In contrast, PowerBi has more collaborative functionalities, which are useful when working in a team (Lombarti, 2022). Both tools support the execution of Python scripts to prepare and manipulate data for visualization. In the case of PowerBi only an installed Python interpreter, and the libraries Matplotlib[23] and Pandas[24] are required. To use Python in Tableau an extension called TabPy[25] is necessary. Similarly, scripts written in the programming language R can be used for data preparation. Another feature supported by Tableau, PowerBi, and other analytic tools is that reports consisting of visualizations can be integrated into websites. For example, PowerBi provides an option to embed a report in a website or portal via an HTML code snippet or a URL (Pawlowski, 2019).

Other analytic tools are MicroStrategy[26], Qlik Sense[27], Oracle Analytics Cloud[28], Sisense[29] and many more. Kaur (2017) describes the difference between these platforms to Tableau. Similarly, Scott (2017) looks at 16 Tableau alternatives without giving any recommendations for one product. Another comparison is done by Scheiner (2021) who lists 15 different analytic tools and also gives information about the individual pricing policies.

[22]https://powerbi.microsoft.com/de-at/
[23]https://matplotlib.org/
[24]https://pandas.pydata.org/
[25]https://github.com/tableau/TabPy
[26]https://www.microstrategy.com/en/business-intelligence
[27]https://www.qlik.com/us/
[28]https://www.oracle.com/business-analytics/analytics-platform/
[29]https://www.sisense.com/

2.7. Summary

Different types of game review platforms exist, where the main difference concerns the types of reviews they provide. While some platforms contain reviews from professional critics, others list reviews from general users. Metacritic and OpenCritic are two platforms which aside from listing user reviews, further aggregate professional critics by calculating an overall professional score. Data from the game distribution platform Steam and the review platform Metacritic have been exploited by researchers for various reasons and to achieve different goals. A summary of the previously discussed studies is shown in Table 2.2. It gives a short overview of the topic of the individual papers, the type of collected data, and from which platform the data has been gathered. It must be noted that, if authors collected reviews, they also collected the associated review metadata, which consists of values like the creation date/time, author information (e.g., playtime at creation), helpfulness votes, and so on. On the other hand, user data corresponds to aspects like friend relationships or what games the users own. Concerning Steam, there are several potential sources from where the data can be collected. The five major sources that could be identified are the Steam website itself, Steam API, Steam Spy, Steam DB, and the database of O'Neill. These sources differ in the intervals the content is updated, the information they provide, the access policy, and the way the information can be accessed. For storing the collected data, the preferred choice is simple JSON files, although studies like Toy et al. (2018) use a relational database. Throughout the chapter, it could be observed that authors collect data using different or redundant approaches. Building a common system might save authors a lot of time and ease the process of getting started. Furthermore, a system providing simple analysis capabilities with the possibility to implement extensions might be useful in terms of reusage and clears the way for easy exchange of ideas between researchers.

Paper	**Topic**	**Data**			**Platform**		
		Reviews	**User Data**	**Game Data**	**Steam**	**Metacritic**	**VR Platform**
(Strååt et al., 2017)	review sentiment	✓	✓	✗	✗	✓	✗
(Kasper et al., 2019)	review helpfulness	✓	✓	✗	✗	✓	✗
(Park & Byun, 2016)	Metascore vs user score	✓	✓	✗	✓	✓	✗
(Sherrick & Schmierbach, 2016)	video game success	✓	✓	✗	✗	✓	✗
(Eberhard et al., 2018)	review helpfulness	✓	✓	✗	✓	✗	✗
(Barbosa et al., 2016)	review helpfulness	✓	✓	✗	✓	✗	✗
(Kang et al., 2017)	review helpfulness	✓	✓	✗	✓	✗	✗
(Ji, 2019)	review sentiment	✓	✓	✗	✓	✗	✗
(Bais et al., 2017)	review sentiment	✓	✓	✗	✓	✗	✗
(Wang et al., 2020)	recommendation system	✗	✓	✓	✓	✗	✗
(Kamal et al., 2020)	recommendation system	✗	✓	✗	✓	✗	✗
(Lin et al., 2018)	early access games	✗	✗	✓	✓	✗	✗
(Windleharth et al., 2016)	genres & tags	✗	✗	✓	✓	✗	✗
(Toy et al., 2018)	popularity	✗	✗	✓	✓	✗	✗
(Li et al., 2021)	playability	✗	✓	✓	✓	✗	✗
(Radianti et al., 2020)	VR education	✗	✗	✓	✓	✗	✓
(Smutny et al., 2019)	VR education	✗	✗	✓	✗	✗	✓
(Fagernäs et al., 2021)	VR relaxation experiences	✓	✗	✓	✗	✗	✓
(Foxman et al., 2020)	VR genre distribution	✗	✗	✓	✓	✗	✗
(Epp et al., 2021)	VR game characteristics	✓	✗	✓	✓	✗	✗

Table 2.2.: Overview of studies exploiting data from the platforms Steam, Metacritic, or VR game distribution platforms like Oculus Store.

3. Design & Concept

This chapter acts as the starting point of our implementation. It starts by discussing the motivation and necessity of our proposed tool. Moreover, it defines functional and non-functional requirements that should be satisfied. Throughout the chapter, different design choices are made and are justified. As a good implementation always starts with a good plan, we further provide a logical architecture for the main components of the tool and suggest a user interface design that is built on top of it. As a last point, we look at two important design patterns used in the implementation.

3.1. Starting Point & Motivation

In the background section, we had a look at a set of studies that explore and analyze data from the Steam platform. An investigation of these studies in terms of data retrieval revealed a redundant implementation for collecting data from Steam.

For saving researchers time and effort, we propose a tool that handles the retrieval, storing, and reading of the data. To make it easier to get started with a project and to get an overview of the collected data, we further provide a set of already implemented analysis tasks in the form of charts.

As most of the previously described studies introduce a new approach in a specific field or aim to analyze certain aspects of the data, we allow the users to apply custom SQL queries on the datasets and to pipe the query results to charts or tables directly in the tool. To overcome the limitations of the SQL query language, the result from an SQL query can be modified using a programming language. This will empower the users to perform advanced analysis tasks including natural language processing, sentiment analysis, or any other machine language approach.

The capabilities of the tool might still be not enough or reach its limits in some cases. To avoid any restrictions, the user should have complete control over the data. Therefore, we enable the user to copy the whole dataset for the usage in other tools. Furthermore, it should be possible to store the query results in form of CSV files and charts in form of images.

To use the functionalities of the proposed tool, some experiences and skills are required. Therefore, the first step is to define the user target group we want to address with our application and who should be able to use our tool with ease.

3.2. User Target Group

The target group of our implemented tool consists of game researchers and game developers. To perform simple analysis, like showing the number of genres or the number of releases over years for a collected dataset, requires only minor knowledge as the tool has a predefined set of analysis tasks that output charts. However, advanced features in the form of a custom analysis have the requisite of understanding and writing SQL queries and Python code, and therefore coding skills are required.

3.3. Requirement Analysis

During the design process of an application, it is substantial to define what the system should do and what quality attributes should be satisfied. Functional requirements define the methods a system should perform, i.e., the expected output and behavior on a specific input. On the other hand, non-functional requirements refer to the quality and performance attributes a software system should meet (Martin, 2021). In this section, functional and non-functional requirements are defined for the main software components of our tool. Hereby, aspects from research done in Chapter 2, are taken into consideration.

3.3.1. Functional Requirements

The main functional requirements for the proposed tool are separated into five components, namely "Collecting General Game Data", "Collecting Reviews", "General Analysis", "Custom Analysis", and "Manage" where each component corresponds to a view described in Section 3.8.

1. **Collecting General Game Data**
   a) Selection between crawling using a Steam search URL or an in-tool configurator.
   b) The tool should allow users to narrow the games to crawl by:
      i. Tag (e.g., Indie, Action, Adventure)
      ii. Type (e.g., Games, Software, Demos, Soundtracks)
      iii. Operating system (Windows, macOS, Linux)
      iv. Virtual Reality support with filters for headset (Valve Index, HTC Vive, Oculus Rift, Windows Mixed Reality), input device (Tracked Motion Controllers, Gamepad, Keyboard / Mouse), play area (Seated, Standing, Room-Scale)
      v. Feature (e.g., Remote Play on Tablet, Played with Steam Controller)
   c) If the in-tool configurator is selected, the system should show all available game types, supported operating systems, Virtual Reality support attributes, and game features to narrow the games to be collected.
   d) The definition of a user-defined project name should be possible.
   e) The project name should be validated to be unique and don't contain any special characters.
2. **Collecting Reviews**
   a) The system should provide a list of all available projects.
   b) Narrowing the reviews by language and purchase type (Steam purchase, non-Steam purchase) should be possible.

c) Collected review data must be stored in the database and be associable with its corresponding game and project.
d) The system should allow the retrieval of reviews written later than a user-defined date.

3. **General Analysis**
   a) The system should provide a list of all available projects.
   b) The user should be able to select the set of analysis tasks to run. A set consists of several related tasks, where each outputs a chart. There are four sets (general analysis set, genre set, tag set, game details set).
   c) The system should allow to only take specific genres, tags, or game details into account.
   d) The system should provide the users all available genres, game details, and tags available for a specific project for restricting the analysis.
   e) It should be possible to save charts to the file system in a common image format.
   f) The system should support the visualization of histograms, bar charts, line charts, grouped bar charts, and tables.
4. **Custom Analysis**
   a) It should be possible to save and open a custom-defined analysis.
   b) The user should be able to define a custom analysis tasks for different chart types and tables.
   c) The system should support the visualization of histograms, bar charts, line charts, grouped bar charts, and tables.
   d) The system should allow the definition of a custom SQL query, where the result is piped to a chart of the selected type.
   e) The system must validate the SQL query result to satisfy the input requirements of the selected chart type (e.g., # of columns, column data types).
   f) The user must be able to define chart-specific attributes for an analysis task (e.g., x-label, y-label, title, corresponding SQL query, Python pre-processing code, label orientation).
   g) It should be possible to save the resulting charts and tables to the file system.

   h) In the case of a table output, it should be possible to copy areas and insert them into an Excel sheet.
   i) The system should notify the user of errors made in the SQL query or Python code.

5. **Manage**
   a) The deletion of projects and their associated data should be possible.
   b) An overview of the general game and review data should be provided per project.
   c) It should be possible to delete the reviews of a certain game id and to re-crawl them.
   d) It should be possible to delete all reviews of a project without deleting the general game information.

3.3.2. Non-Functional Requirements

In contrast to the functional requirements, we do not split the definition into several components but provide a general overview of the non-functional requirements we hope to achieve and which we keep in mind during the development process of the tool.

1. **Performance**
   While the tool is executing a task, the user interface should not block user input.
2. **Reusability**
   A created custom analysis should be reusable.
3. **Scalability**
   The amount of collected data should only be limited by the capacity of the hard drive or the maximum size of the used SQL database.
4. **Maintainability**
   Deleting data (projects, reviews) from the database should be possible and easy to execute.
5. **Portability**
   a) The tool should be easy to install on a Windows PC.

   b) The exchange of Steam data and custom analysis scripts between different users of a tool should be possible.

6. **Usability**
   Collecting general game and review data, as well as running the general analysis should be easy and self-explanatory.
7. **Extensibility**
   a) New chart types for performing a custom analysis in the program code should be easy to add by other developers.
   b) Crawling additional information (values) from the web pages of the individual games should be easy to implement.
8. **Configurability**
   It should be configurable for which games data should be crawled from Steam.
9. **Availability**
   The tool should be usable even if the device the tool runs on has no internet connection.
10. **Recoverability**
    In the case of an error, the tool should not crash but inform the user about the type of error, and if possible, how to solve it. This is especially important regarding the custom analysis. Here, invalid Python code or SQL select statements should not lead to a crash but provide the user a meaningful description of the error.
11. **Awareness**
    While a task is executed, the user should be aware of the ongoing process (e.g., loading bar)
12. **Consistency**
    The design of the tool in terms of color usage, font, and layout should be consistent.

3.4. Programming Language

Python currently belongs to the most popular programming languages, where the PYPL index even lists Python as the most popular one (PYPL, 2021). The index is determined by analyzing how often language-specific tutorials are searched via the Google search engine. For the calculation, data is gathered from Google Trends. In addition, Python takes first place in IEEE Spectrum's top programming languages ranking of the year 2021, where the ranking is created by weighting metrics from different sources (Cass, 2021).

According to different websites (D. Costa, 2020b; Gallinelli, 2021; Muthyala, 2021), Python is the language number one for data science and machine learning purposes. The main advantage of Python is seen in the numerous available data science tools for analysis, visualization, and data collection. Furthermore, Python is well documented, has a large user community for support, and is easy to learn.

The usage of Python for analyzing and collecting Steam data in studies described in Section 2.3 and Section 2.4 supports the popularity of Python for data science. The studies utilize different Python libraries depending on the task. For natural language processing, the Python library NLTK[1] is popular amongst the authors (Eberhard et al., 2018; Kamal et al., 2020; Kasper et al., 2019). The API provides a set of text processing tools for processing text, categorizing and tagging words, classifying text, extracting information from text, and analyzing sentence structures. It is completely open-source and driven by a big community (Bird et al., 2009). For machine learning purposes, Scikit-learn[2] is used by authors like Ji (2019) and Kamal et al. (2020). The library provides a large number of algorithms for classification, regression, clustering, dimensionality reduction, model selection, and data pre-processing. Other libraries which are used are Numpy[3], gensim[4], and Pandas[5]. Numpy is a package for scientific computing and is laid out for performance. Gensim is used for topic modeling, and Pandas is a data analysis and manipulation tool.

Due to the libraries available for Python concerning data science and visualization, Python is our programming language of choice. Furthermore, we argue this decision that Python is easy to learn and use. As we allow users

[1]https://www.nltk.org/
[2]https://scikit-learn.org/stable/
[3]https://numpy.org/
[4]https://radimrehurek.com/gensim/
[5]https://pandas.pydata.org/

of our tool to create a custom analysis through self-written scripts, it makes sense to choose an easy programming language. In addition, it is easier to run Python scripts in Python rather than in another programming language like C++, where it is possible to run Python code using the Python C API (Python Software Foundation, 2021).

Python being the programming language of our choice, we look at the data storage options it supports and which one we want to utilize.

3.5. Data Storage

With Python, it is easy to read files that have different encodings like JSON or XML. As previously mentioned in the background section, different studies use simple JSON encoded files to store their collected Steam data. However, in our case, storing Steam data in simple JSON files is not well suited. The reason for this is that we usually do not work on the whole data, but only on a small portion depending on the performed data requests. In the case of JSON files, usually the whole file is read into memory, although only a small part of the file is needed. To reduce the amount of unnecessary loaded data, the implementation of an advanced file structure and additional file indexing might help. However, the realization of such structures is time expensive and avoidable as there are other data storage approaches like Relation Database Management Systems (RDBMS) that already provide such features. RDBMS are databases that store data in a structured way using rows and columns. Such systems aim to make the data accesses fast and efficient via advanced indexing. The Structured Query Language (SQL) allows the definition of complex queries, where it is easy to request relations between different pieces of data.

As performing complex queries on our Steam data is an essential part of our tool, we prefer RDBMS over simple JSON files. However, we want to avoid a complex setup of the database and do not want a database that is shared among different people. Furthermore, the database should be delivered as part of the tool. A database system that is local, but still provides the SQL syntax is SQLite. In contrast to online databases like MySQL, SQLite can not handle multiple users, is less scalable, and has fewer security features.

However, SQLite is easier portable as it only consists of a single file (Edward, 2021).

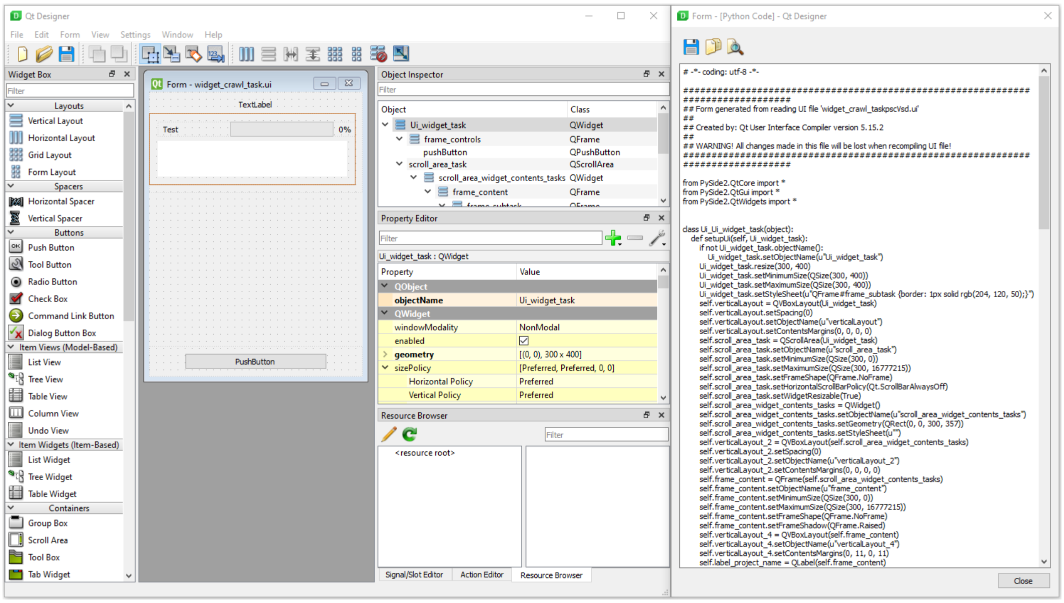
3.6. User Interface Framework

Having the programming language and data storage fixed, we are looking for the Graphical User Interface (GUI) framework that best fits our needs. For the decision, we determine the five most popular frameworks based on web entries in the form of posts and blogs providing a ranking (AskPython, 2021; D. Costa, 2020a; Fatima, 2017; Nederkoorn, 2021). As each entry has a different ranking, we attribute ten points to the first place and reduce the points by one for each consecutive one. We perform this process for each source and sum up the points for each framework. By considering only the five frameworks with the most points results in the ranking: PyQt5[6] > Tkinter[7] > Kivy[8] > wxPython[9] > PySide2[10].

3.6.1. PyQt5

PyQt5, developed by Riverbank Computing Ltd., is built around the Qt framework, which is known as one of the most popular cross-platform GUI frameworks. One very convenient feature of Qt is that layouts and widgets can be created via drag and drop, or programmatically. Furthermore, a layout that is built with the drag and drop designer can be converted to C++ or Python code. An example of a widget created with Qt Designer can be seen in Figure 3.1. The left window shows the designer tool and the right window shows the Python code that corresponds to the created design. The Python code is compatible with PySide2 and PyQt5 (with minor changes). The installation of PyQt5 can be achieved via the command "pip install pyqt5". PyQt5 only requires one code base for several platforms, including

[6]https://riverbankcomputing.com/software/pyqt/intro
[7]https://docs.python.org/3/library/tkinter.html
[8]https://kivy.org/
[9]https://www.wxpython.org/
[10]https://pypi.org/project/PySide2/



```
# -*- coding: utf-8 -*-

################################################################################
## Form generated from reading UI file 'widget_crawl_taskpscVsd.ui'
##
## Created by: Qt User Interface Compiler version 5.15.2
##
## WARNING! All changes made in this file will be lost when recompiling UI file!
################################################################################

from PySide2.QtCore import *
from PySide2.QtGui import *
from PySide2.QtWidgets import *


class Ui_Ui_widget_task(object):
    def setupUi(self, Ui_widget_task):
        if not Ui_widget_task.objectName():
            Ui_widget_task.setObjectName(u"Ui_widget_task")
        Ui_widget_task.resize(300, 400)
        Ui_widget_task.setMinimumSize(QSize(300, 400))
        Ui_widget_task.setMaximumSize(QSize(300, 400))
        Ui_widget_task.setStyleSheet(u"QFrame#frame_subtask {border: 1px solid rgb(204, 120, 50);}")
        self.verticalLayout = QVBoxLayout(Ui_widget_task)
        self.verticalLayout.setSpacing(0)
        self.verticalLayout.setObjectName(u"verticalLayout")
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.scroll_area_task = QScrollArea(Ui_widget_task)
        self.scroll_area_task.setObjectName(u"scroll_area_task")
        self.scroll_area_task.setMinimumSize(QSize(300, 0))
        self.scroll_area_task.setMaximumSize(QSize(300, 16777215))
        self.scroll_area_task.setFrameShape(QFrame.NoFrame)
        self.scroll_area_task.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.scroll_area_task.setWidgetResizable(True)
        self.scroll_area_widget_contents_tasks = QWidget()
        self.scroll_area_widget_contents_tasks.setObjectName(u"scroll_area_widget_contents_tasks")
        self.scroll_area_widget_contents_tasks.setGeometry(QRect(0, 0, 300, 357))
        self.scroll_area_widget_contents_tasks.setStyleSheet(u"")
        self.verticalLayout_2 = QVBoxLayout(self.scroll_area_widget_contents_tasks)
        self.verticalLayout_2.setSpacing(0)
        self.verticalLayout_2.setObjectName(u"verticalLayout_2")
        self.verticalLayout_2.setContentsMargins(0, 0, 0, 0)
        self.frame_content = QFrame(self.scroll_area_widget_contents_tasks)
        self.frame_content.setObjectName(u"frame_content")
        self.frame_content.setMinimumSize(QSize(300, 0))
        self.frame_content.setMaximumSize(QSize(300, 16777215))
        self.frame_content.setFrameShape(QFrame.NoFrame)
        self.frame_content.setFrameShadow(QFrame.Raised)
        self.verticalLayout_4 = QVBoxLayout(self.frame_content)
        self.verticalLayout_4.setObjectName(u"verticalLayout_4")
        self.verticalLayout_4.setContentsMargins(0, 11, 0, 11)
        self.label_project_name = QLabel(self.frame_content)
```

Figure 3.1.: Example of a widget created with Qt Designer (left) with the corresponding Python code for PySide2 (right).

Windows, Mac, Android, and Linux. In addition, it can be used with GPL or with a commercial license. In the former case, the end-user must have unlimited access to the source code of the application and must have the rights to modify, share and execute the code. Otherwise, a commercial license is necessary (AskPython, 2021; D. Costa, 2020a; Fitzpatrick, 2020; Nederkoorn, 2021).

3.6.2. Tkinter

The main advantage of Tkinter is that it is directly shipped with the standard Python3 installation and can therefore be used without installing any further packages. It provides a wide range of widgets, including buttons, radiobuttons, checkboxes, sliders, labels, and text fields. Furthermore, it provides a file dialog for opening and saving files and a canvas widget for drawing custom shapes (AskPython, 2021; D. Costa, 2020a; Fatima, 2017; Nederkoorn, 2021).

3.6.3. Kivy

Kivy is a community project and can be used for free under the MIT license since version 1.7.2. It supports over 20 different widgets that can be extended. For performance reasons, the toolkit is partly written in C using Cython and is GPU accelerated via the graphics pipeline OpenGL ES 2. Kivy supports the operating systems Windows, Android, iOs, and Raspberry Pi. Hereby, the same code can be compiled for the different operating systems without any changes. A feature of Kivy is that it supports inputs from various devices out of the box, which eases the development of multi-touch applications. To support Kivy, individual persons and companies can donate money and will be listed on their website (Kivy Organization, 2021; Nederkoorn, 2021).

3.6.4. wxPython

Like Kivy, wxPython is an open-source, free to use and cross-platform GUI toolkit. The library wraps the in C++ written wxWidgets cross-platform library. It provides a natural feeling look by using the native widgets of the individual operating systems. An ongoing project of the wxPython developers is called Phoenix that aims to improve the wxPython library in terms of speed, extensibility, and maintainability. Phoenix is not fully compatible with the classic wxPython library, but the migration only requires minor or even no changes at all (D. Costa, 2020a; wxPython Team, 2020).

3.6.5. PySide2

PySide2 is very similar to PyQt5, with the difference that PySide2 was developed by Qt and is available under the LGPL license. Both frameworks wrap the Qt library, and thus, a large part of their APIs overlap. In contrast to the GPL license, the LGPL license does not force the developer to provide the source code unless changes are made directly to PySide2 itself (Fitzpatrick, 2020).

PyQt5 takes first place in our calculated ranking. Furthermore, we would not run into any problems concerning the licensing as we want to distribute our source code along with the tool. In fact, given access to the source code is in our favor, as it enables the users to write extensions on their own and fit the tool to their needs. Moreover, when reading through the PyQt5 documentation of Fitzpatrick (2020), we realized that PyQt5 supports all the user interface features that might be needed for our implementation. Thus, PyQt5 is the user interface framework of our choice.

Now that we have defined the main components our tool will be built upon, we require a concept that defines the logic of our proposed tool in a simplified manner. The intention is to define a rough structure in terms of process flow and design to ease the implementation and give it a direction. This is done in the next section, where a logical architecture is defined for the main parts of the tool.

3.7. Logical Architecture

To get a high-level overview of our tool, we have a look at the logical architecture for collecting and analyzing data from the Steam platform. The aim is to give first insights on how the overall system works and is not to provide details about the implementation. Aspects and features which do not contribute to the overall understanding of the tool are not discussed as part of this section.

In general, the tool can be split into two main parts, data collection and data analysis. As these two parts can be split well, we discuss them separately. What both parts have in common is that they have access to the same local SQLite database. While the data collection process mainly inserts data, the data analysis process mainly queries data from the local database. However, the deletion of already available data is not handled by either of them. Therefore, we provide a logical architecture for managing the data in Subsection 3.7.3.

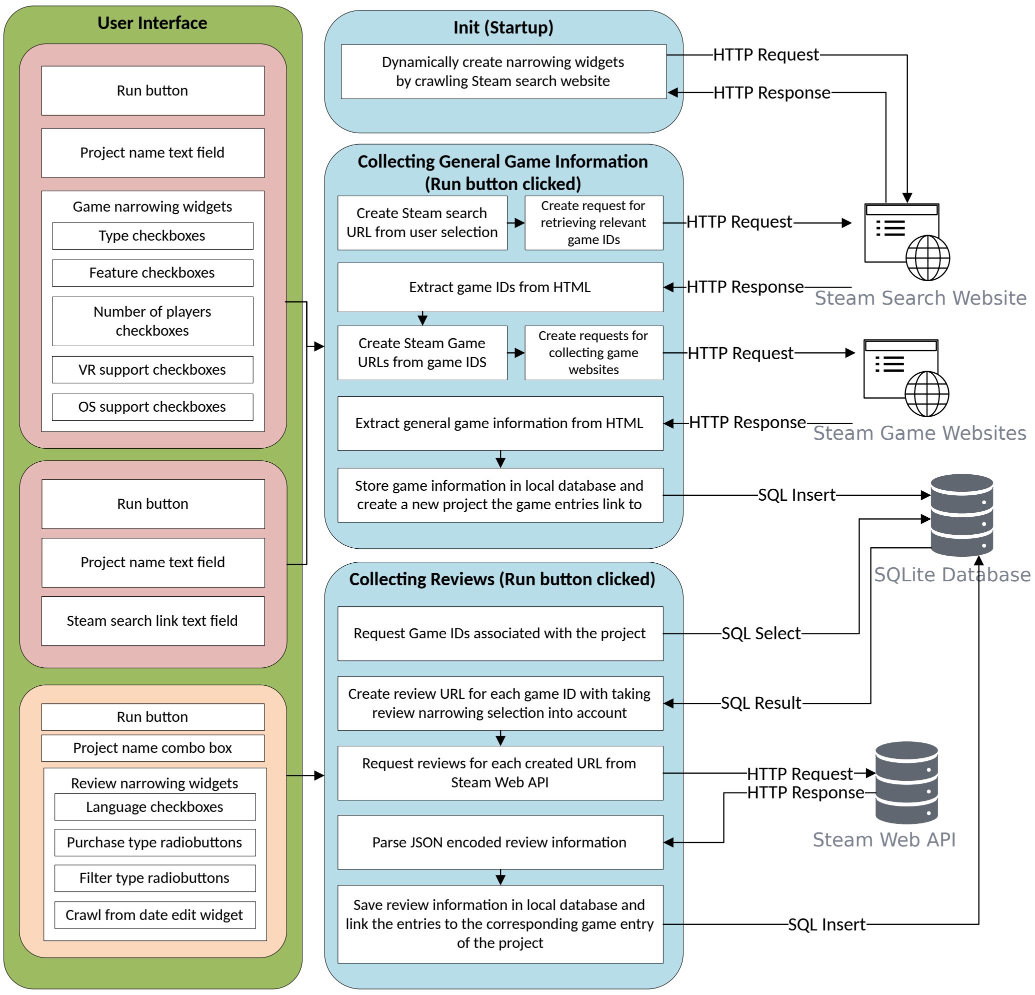

User Interface
Run button
Project name text field
Game narrowing widgets
Type checkboxes
Feature checkboxes
Number of players checkboxes
VR support checkboxes
OS support checkboxes
Run button
Project name text field
Steam search link text field
Run button
Project name combo box
Review narrowing widgets
Language checkboxes
Purchase type radiobuttons
Filter type radiobuttons
Crawl from date edit widget
Init (Startup)
Dynamically create narrowing widgets by crawling Steam search website
HTTP Request
HTTP Response
Collecting General Game Information (Run button clicked)
Create Steam search URL from user selection
Create request for retrieving relevant game IDs
HTTP Request
Extract game IDs from HTML
HTTP Response
Steam Search Website
Create Steam Game URLs from game IDS
Create requests for collecting game websites
HTTP Request
Extract general game information from HTML
HTTP Response
Steam Game Websites
Store game information in local database and create a new project the game entries link to
SQL Insert
SQLite Database
Collecting Reviews (Run button clicked)
Request Game IDs associated with the project
SQL Select
Create review URL for each game ID with taking review narrowing selection into account
SQL Result
Request reviews for each created URL from Steam Web API
HTTP Request
HTTP Response
Steam Web API
Parse JSON encoded review information
Save review information in local database and link the entries to the corresponding game entry of the project
SQL Insert

Figure 3.2.: High-level overview of the information collection process.

3.7.1. Data Collection

As most studies aim to analyze a subset of the available games on Steam, we need to find a way to narrow the games to crawl. Steam already provides a page to search for specific games which fulfill certain criteria. The big advantage of using the search results from Steam is that we do not need a list of all available games beforehand to restrict the results. Instead, we can make use of the search page and let Steam do the work for us. Our tool allows two different possibilities to build a request for crawling general game data, an in-tool configurator, or by directly using the Steam search URL. In the former case, we provide the user with a set of widgets for narrowing the games. These widgets are dynamically created at the startup of the tool and mirror the narrowing options of the Steam search page. When a crawling process is started, the Steam search URL is created taking the selection of the user into account. Optionally, the user can directly provide the Steam search URL which results from the configuration on the Steam search page itself.

The Steam search URL, with its narrowing parameters, is used to obtain a list of relevant game IDs. How this is achieved in detail is described in Chapter 4. These game IDs are then used to build the Steam game URLs of the form *https://store.steampowered.com/app/xxxxxxx/*, where the xxxxxxx is replaced with the game ID. For each of the created URLs, the corresponding HTML is retrieved via an HTTP GET request. Game-related information like the publisher, developer, release date, associated genres, user-defined tags, rating, and the title, is then extracted from the HTML files and stored in a local SQLite database. However, no review data is extracted from the individual game pages.

To crawl general game data, the user has to provide a unique project name. Hereby, each collected game entry in the SQLite database is linked to the newly created project entry. The idea behind this is to provide a project-like structure, where a project is associated with a crawling process for collecting general game data.

As can be seen in Figure 3.2, the processes for collecting general game data and reviews are handled separately. The intention behind this is that depending on the data requirements of a user, reviews might not be required.

Furthermore, to be able to request the reviews, the game IDs are required beforehand. The reason for this is that for obtaining the reviews, no custom webcrawler is written. Instead, the Steam Web API method for collecting reviews is utilized, which requires the game ID to collect the reviews of a game. The reviews are always collected per project. Hereby, the user can select the project from a drop-down list of the already existing ones. The reviews can further be narrowed by language, and purchase type (all, Steam purchase, non-Steam purchase). This is realized via checkboxes and radiobuttons. Furthermore, the Steam Web API provides the possibility to neglect reviews before a specific date. Therefore, a date edit widget is added to the user interface for selecting the start date. When a process for collecting the reviews is started, the associated game IDs of the selected project are requested from the local SQLite database. For each of the game IDs, a URL is built where the request parameters are set accordingly depending on the user settings. These URLs are used to request the reviews from the Steam Web API, where the reviews are returned in JSON format by default. Per response, the JSON encoded files are parsed and the contained information is saved in the local database. Hereby, each review entry is linked to the corresponding game entry of the project. The tool also keeps track of which game entries the reviews have already been collected. This allows to pause the collection process when closing the application and to continue it at a later point.

The collected general game data and the optionally or partially available review data that is stored in the local database can be analyzed using a set of predefined analysis tasks or via custom-defined ones. For this purpose, we provide a logical architecture in the next part.

3.7.2. Data Analysis

Figure 3.3 shows the concept for running predefined analysis tasks on the collected data. As can be seen, there is a combo box (drop-down list) for selecting the project of which the data should be considered. On the change of a project, all available genres, tags, and game details that are assigned to at least one of the project's games are requested from the local database and are visualized via checkboxes in the user interface. These widgets

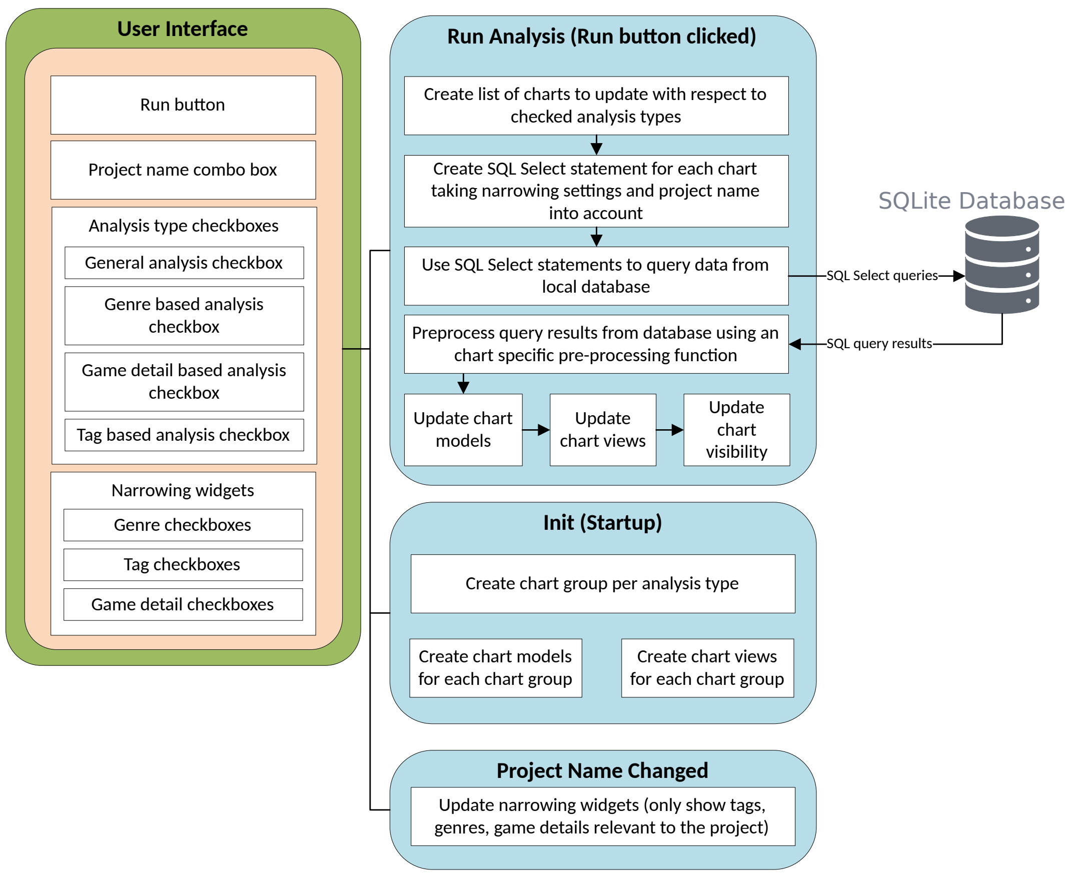

User Interface
Run button
Project name combo box
Analysis type checkboxes
General analysis checkbox
Genre based analysis checkbox
Game detail based analysis checkbox
Tag based analysis checkbox
Narrowing widgets
Genre checkboxes
Tag checkboxes
Game detail checkboxes
Run Analysis (Run button clicked)
Create list of charts to update with respect to checked analysis types
Create SQL Select statement for each chart taking narrowing settings and project name into account
Use SQL Select statements to query data from local database
Preprocess query results from database using an chart specific pre-processing function
Update chart models
Update chart views
Update chart visibility
SQLite Database
SQL Select queries
SQL query results
Init (Startup)
Create chart group per analysis type
Create chart models for each chart group
Create chart views for each chart group
Project Name Changed
Update narrowing widgets (only show tags, genres, game details relevant to the project)

Figure 3.3.: High-level overview of the general game data analysis process.

are utilized for restricting the analysis tasks to games that have certain attributes. Each task is associated with a chart (e.g., bar chart, line chart), which is directly visualized in the tool. The user has the option to select four different types of analysis - general analysis, genre-based analysis, game detail-based analysis, tag-based analysis. In the case of the general analysis, the narrowing widgets are not considered. In the case of the other three types, only the respective narrowing widgets are taken into account (e.g., genre-based analysis - genre checkboxes) except for a few charts where genres, tags, and game details are set into relation with each other.

When the analysis process is started, the system dynamically creates an SQL Select statement per chart based on the project name and the checked narrowing checkboxes. The query results are pre-processed using a chart-specific pre-processing function before they are stored in a chart model. The individual views of the charts are then updated using the data from the models.

The motivation behind the general analysis is to allow the user to obtain a quick overall overview of the project and the collected dataset. However, in some cases, this might not be enough. Therefore, the user has the possibility to create a custom analysis. The logical architecture is similar to the logical architecture of the general game data analysis process with the difference that the SQL Select statements and other attributes are not pre-defined. The user can add and configure a custom task in a separate dialog. The configuration provides different chart types via a drop-down list. Depending on the selected chart type, different attributes, like x-label, y-label, and title, are configurable. However, what all the chart type configurations have in common is that they allow the definition of a SQL Select statement and an optional Python pre-processing script. As the different chart types have different data requirements, the SQL Select statements have to fulfill specific criteria in terms of column count and data types. In the case, the Python script is enabled, only the resulting data from the script has to be of a specific format. Hereby, the script has access to the query result of the SQL Select statement. After a configuration is applied, the task is added to the task list. Furthermore, the task is visualized via a button in the user interface for later modifications. When the custom analysis is started, all the created tasks are executed and the corresponding charts are created and visualized.

So far we only looked at cases, where data is either inserted or read from the local SQLite database. However, the user might want to delete data from the database as well. A reason for deleting a project or its reviews might be a wrong narrowing configuration during the review collection process or to free hard disk memory. Therefore, we provide a third functional architecture for managing the projects and the associated game and review data in the following section.

3.7.3. Data Management

The basic functional concept for managing the data stored in the SQLite database is quite simple. The user interface provides a combo box (drop-down list) for selecting an already available project. When a project is selected, the corresponding general game data configuration and if available, the review collection configuration, are displayed. Hereby, the idea is to provide the user an overview of when the project has been created and which narrowing attributes have been used during the collection process. Furthermore, the user interface contains buttons for deleting the reviews or the whole project. For each deletion request, a SQL Delete statement is dynamically created based on the project selection. If only the reviews of a project are deleted, it is possible to collect them again using the same or another narrowing configuration.

The suggested local architectures for the main parts of the tool mention different user interface elements and for which purpose these elements are used. However, the arrangement and integration of the widgets in the user interface itself are still unknown. As a result, the next section proposes a user interface design that aims to offer a clean separation between the main components of the tool.

3.8. Tool Design

The user interface design consists of six different views. These can be switched via a menu which is on the left side of the window as can be

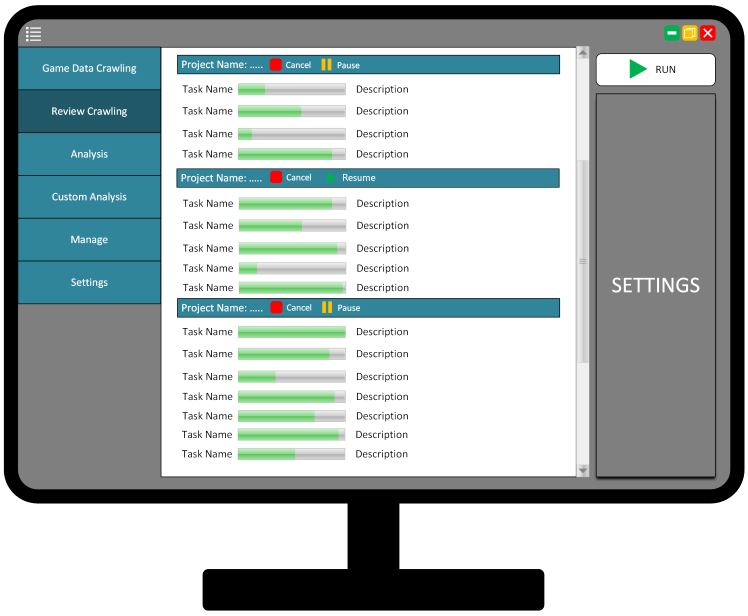


Figure 3.4.: Mock-up of the views for collecting general game data and reviews.

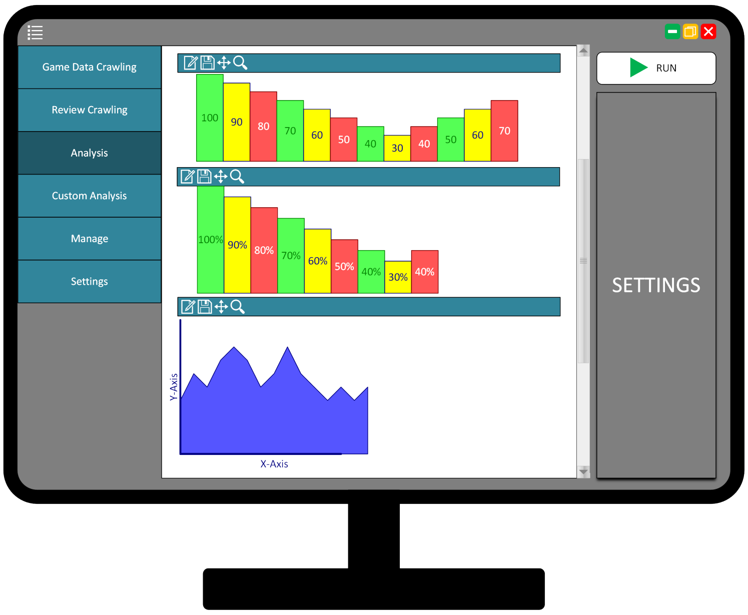


Figure 3.5.: Mock-up of the view for performing the general analysis.

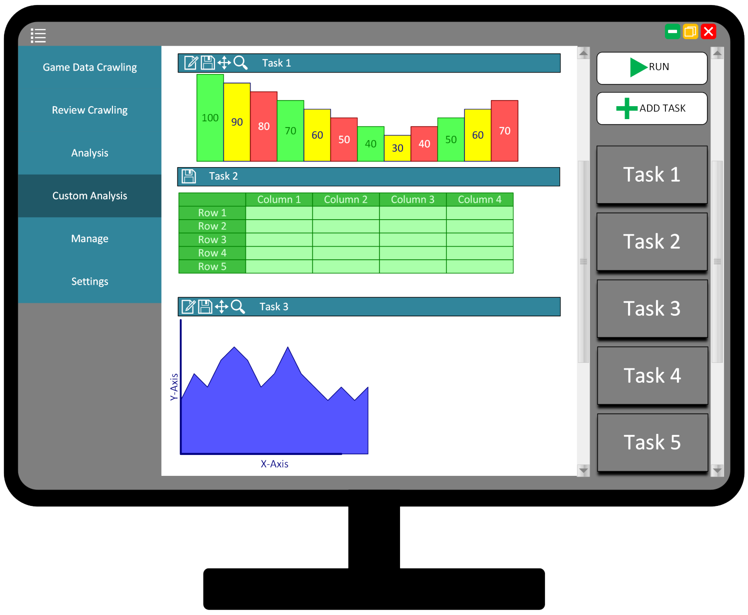


Figure 3.6.: Mock-up of the view for performing a custom analysis.

seen in Figure 3.4. Hereby, the currently selected view is highlighted via a different color. Furthermore, a button in the upper left corner allows collapsing the menu to increase the working area of the selected view. The layout of the individual views is split into a content frame and a right sidebar. While the content frame shows the results or progress of tasks, the right sidebar encompasses the widgets for configuring and customizing the tasks.

As we previously defined a clear separation for collecting general game data and for collecting reviews, the user interface provides two separate views as well. The layouts of the views (illustrated in Figure 3.4) are similar and only differ in the encapsulated widgets of the settings frame. The content frame shows the progress of the executed tasks. Each task consists of one or more subtasks and has its own taskbar, which contains the corresponding project name and controls for manipulating the execution state, i.e., buttons for canceling, pausing, or resuming a task.

Figure 3.5 shows the layout for performing the general analysis. The settings frame contains all the in the previous section mentioned widgets that are required for the configuration. This includes a drop-down list for selecting the project, the widgets for selecting the sets of analysis tasks that should be executed, and the narrowing widgets for restricting the data to specific genres, tags, or game details. On the other hand, the content frame shows the resulting charts. Each chart has a toolbar that enables the user to make adjustments or save the chart to an image file.

The content frame of the custom analysis view, visualized in Figure 3.6, is the same as the one from the general analysis. However, the right sidebar differs as it additionally provides a button for adding custom tasks and does not contain any narrowing widgets. When a task is added, it is listed in form of a button in the sidebar. As described in the previous section, a click on a task button would open a dialog window for adjusting the task.

The views which remain are the manage window and the settings window. The right sidebar of the manage window contains the buttons for deleting the project or its reviews and a drop-down list for selecting the project an operation should be performed on. The content frame shows the project-related information concerning the collected games and reviews. Compared to the other windows, the settings window does not provide a sidebar as it only contains widgets for general settings.

So far, we have made important choices regarding the programming language, data storage, and UI framework. Additionally, we have defined a logical architecture for the main components of our tool and have proposed a design that is built upon this architecture. Thus, we have a well-defined design and concept, and can finally start with the implementation. However, as the last point of this chapter, we want to discuss design patterns that will be used throughout the whole implementation process and reflect two essential concepts of the implementation.

3.9. Design Patterns

The two most important design patterns, we are going to implement, are the Hierarchical-Model-View-Controller (HMVC) design pattern and the worker thread pattern.

3.9.1. HMVC Design Pattern

The Model-View-Controller (MVC) design pattern consists of three parts:

- The **Model** contains the application data, and the business logic performed on the data. It provides methods for requesting and changing the data. In an object-oriented programming language like Python, a model can be realized with an object encapsulating the data attributes and the business logic methods.
- The **View** visualizes the data of its related model. The view can only read data from the model but does not have the right to change the model's state. Different views can exist for the same model but may look completely different (e.g., different style, UI design).
- The **Controller** handles the communication between the view and model. It listens to signals triggered by either of them and reacts by executing a method or an instruction that works on the model or view. As a result, the controller can tell the view to update itself after a change to a model has been applied.

The idea behind the MVC design pattern is to completely separate the data model and presentation from each other for the purpose of maintainability and code readability. An essential aspect of MVC is that the model does not know anything about the view, and the view can only read data from the model. There are different variations of the MVC design pattern. The model can either notify the view to redraw certain parts as in the original version (Cai et al., 2000) of the MVC pattern directly or notify the controller to trigger an update of the view. In the former case, the observer pattern is necessary, where the view listens to signals from the model. Otherwise, the model would require information about the view, which validates the characteristics of the MVC design pattern (Fitzpatrick, 2020; Krasner, 1988; Nunes, 2020).

In the Hierarchical-MVC (HMVC) design pattern, MVC blocks are arranged in tiers forming parent-child relationships. Such an architecture is well suited for widget-based applications, where widgets are structured in layers as well. For example, a GUI might have a frame that contains several frames of the same type. The HMVC pattern allows to reuse components and eases the communications between parents and children, while still keeping a structure that is well to maintain. If a controller is not able to handle a message or signal from the view, model, or from a child, it can pass it to its parent, which on the other hand, tries to handle the message itself. This is usually achieved via loose coupling, where a parent listens to specific signals from its children (Cai et al., 2000).

Figure 3.7 shows a variation of the HMVC design pattern we decided to use in our implementation. Concerning the individual MVC blocks, we do not use the observer pattern between the model and view. Instead, all communication is handled by the controller. The reason for this is that in some cases, it might be necessary to transmit messages to a parent controller. As this is usually done by the controller, it needs to listen to signals emitted by the model. However, to avoid that both, the view and the controller listen to the signals, which would result in additional performance overhead, we decided to let the controller trigger all view updates. The disadvantage of this approach is that we have a more tight coupling between the controller and view. In the figure, we can observe dashed and continuous lines. While the dashed lines refer to loose coupling, the continuous ones refer to tight coupling. As previously mentioned, loose coupling is achieved via signals. We have loose coupling from a child to its parent, which means that the parent only connects to the signals of its children. As a result, the children are not provided with any information about their parents. However, in the other direction, a tight coupling exists, which indicates that the parent is completely aware of its children and can access their public methods.

3.9.2. Worker Thread Pattern

The worker thread pattern consists of a job queue and a thread pool. Tasks are pushed to the job queue and are polled and run by the threads until there are no more jobs left. The idea behind worker threads is to achieve a higher utilization by parallelism. However, in general, threads in a user interface environment have the advantage that they allow to run "heavy" tasks without blocking the user interface. Thus, a good approach is to utilize threads for any long-running task.

Concerning Python, some caveats have to be considered. The programming

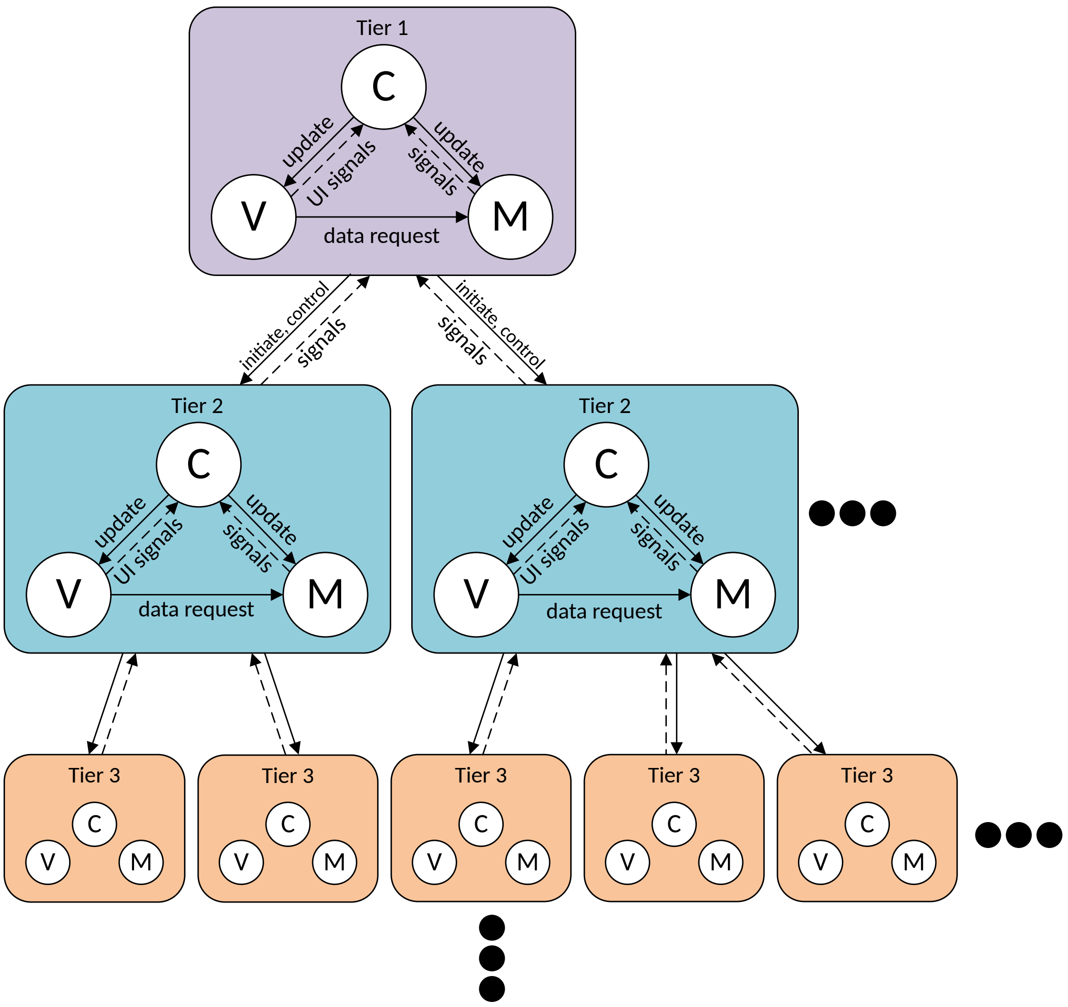


Figure 3.7.: Variation of the Hierarchical-Model-View-Controller design pattern.

language uses a lock known as the Global Interpreter Lock (GIL), which has the effect that only one thread can have control over the interpreter at a time. As a result, only the thread that currently holds the GIL can execute code. Therefore, the GIL has a negative impact on performance in multi-threaded environments. Still, using several threads to work on a task cooperatively makes sense in specific cases. To give other threads execution time, the GIL is released in fixed intervals. In addition, the GIL is released when an input or output request is executed and is reacquired when the request is finished. Input and output requests refer to operations where data is read from or written to a database, network, file, or other external sources. Furthermore, many Python libraries are written in C via C extensions, where the GIL is released in cases where work is done outside the Python API (Ajitsaria, 2017; Fitzpatrick, 2020).

As we have many HTTP requests simultaneously when collecting general game data or reviews, using more than one worker thread is preferable. Likewise, when executing multiple SQL Select statements, several worker threads can run in parallel if the utilized Python C extension releases the GIL. This happens in the case of the Python SQLite module if a separate database connection is opened per thread (Stack overflow, 2012; Willison, 2021).

3.10. Summary

This chapter defined the functional and non-functional requirements of the proposed tool. Based on these requirements and the research done in Chapter 2, several decisions were made. Python was chosen as the programming language of choice for the reason that it is best suited for data science tasks, and is easy to learn. Concerning data storage, a relational database is preferred over JSON encoded files, as RDBMS are more flexible and performant in terms of inserting, queuing, deleting, and updating data. Furthermore, a local database, SQLite, was selected instead of an online one to keep the traffic over the network low and make the data easily portable for the usage in other analytic tools. The user interface framework for Python was selected based on an investigation of the five most prominent ones. The chosen and highest-ranked framework, PyQt5, has all features that are necessary for the implementation. Moreover, it is cross-platform and allows to extend and write custom widgets. However, an outstanding feature is that it allows the drag-drop creation of a layout, which can be directly translated to Python code. After the decisions concerning tools and frameworks were made, a logical architecture for the main components of the tool was discussed, which defines a rough structure and logic flow that should ease the implementation and give it a direction. The logical architecture was split into three parts, namely data collection, data analysis, and data management. Each of these parts represents a certain aspect of the application. Data collection refers to the process of collecting general game data and reviews, data analysis defines the process for analyzing the collected data, and the data management process is responsible for deleting projects and their corresponding reviews. On top of the logical architecture, a user interface design was built, that should provide a clean separation of the discussed aspects. As the last point, the chapter described design patterns and how they work. The next chapter handles the implementation of the defined design and concept.

4. Implementation

This chapter provides deep insights into the implementation based on the functional/non-functional requirements, design, and logical architecture of the previous chapter. It describes the structure of the user interface and how it is given its style. Moreover, it shows the integration of the SQLite database into the project with additional extensions. Furthermore, the chapter introduces the architecture used for collecting Steam data as well as for analyzing the data. In addition, the chapter describes the implementation steps that are necessary for collecting general game data and reviews. Similarly, the implementations for performing the general analysis as well as a custom analysis are described. A further section documents how the result of a performed analysis is handled and visualized in form of different chart types and tables. As the system should be robust to errors, a separate section defines how errors are handled and how they are resolved. The last topic of this chapter details how the tool is prepared for distribution to the end-user.

4.1. User Interface

The user interface (UI) allows users to interact with applications. In the case of our tool, we speak of a graphical user interface (GUI) as the interaction with the system is based on graphical elements like buttons which should be arranged in an intuitive and meaningful way. A user interface consists of one or more layouts that describe the arrangement of the graphical elements.

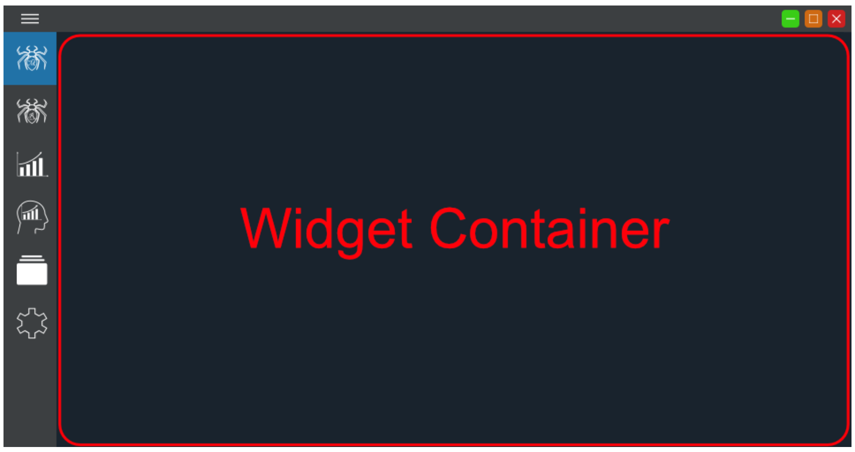


Figure 4.1.: Main view.

4.1.1. Layouts

As previously mentioned, the Qt Designer allows to create layouts (widgets) and to convert them to Python scripts that are compatible with PyQt5. However, one problem is that the created layouts are static. Thus, only static user interface elements that also remain static in the tool are added via Qt Designer. The Python scripts encapsulate the static widgets in a class. To extend a class by dynamic widgets, a derived class is created which inherits the widgets and methods. Optionally, the converted Python scripts and classes can be modified directly. However, this has the drawback that converting a Qt Designer layout to the Python script again would override the changes. Therefore, a clean separation is introduced using inheritance.

The above-mentioned approach was used to create the main view, which can be seen in Figure 4.1. It contains the title bar, a menu sidebar, and a widget container. If a menu button is clicked, the content of the widget container is replaced with the widget of the corresponding view. Like the main view, they all have a separate class that inherits properties from the class created via Qt Designer.

4.1.2. Custom Widgets

Each of the views contains various kinds of widgets, including buttons, checkboxes, textfields, etc. For this, Qt provides a set of default widgets[1], which can be customized and extended. As re-usability is an important concept of our design, a collection of custom widgets is created in form of classes that are derived from default widget classes. An overview of the most relevant custom widgets is given in Table 4.1. It lists the names of the custom widget classes and their parent classes with short descriptions.

4.1.3. Style

By default, all the widgets have a platform-dependent default style. However, it is also possible to use self-defined or third-party themes to give applications a modern and more appealing look. As dark mode themes are considered to reduce eye strain (Erickson et al., 2020; Fitzpatrick, 2020), we use one for our application as well. Instead of defining everything on our own, the Python library QDarkStyle[2] is utilized to give the application its main look. The library provides a style sheet that is compatible with PyQt5. To obtain and set the style sheet in Python, only a few lines of code are required as shown in Listing 4.1. However, in some cases, a separate or an extended style is required. To overwrite or extend parts of the QDarkStyle style sheet, additional .qss (Qt Style Sheets) files are created. Such files define the properties that should be applied to whole widget types, widget objects with a specific attribute, or objects of a derived widget class. Furthermore, the files contain the definitions on how elements should look in the case of certain states (e.g., mouse hover). A reference guide and the available properties can be found on the Qt documentation website[3]. When setting a style in Python for a widget where the style is already set, the overlapping definitions are replaced with the new ones. The style is automatically applied to all child widgets, i.e., the widgets the widget encapsulates, but not to parent widgets.

[1]https://doc.qt.io/qt-5/gallery.html
[2]https://pypi.org/project/QDarkStyle/
[3]https://doc.qt.io/qt-5/stylesheet-reference.html

Class (Parent Class)	**Description**
LoadingDialog (QDialog)	Dialog which animates a throbber (loading icon) with an optional text. It has a translucent background and provides a function for closing the dialog.
LoadingFrame (QQFrame)	Frame which contains an animated throbber. It has functions for starting and stopping the animation.
QHLine (QFrame)	Horizontal separation line.
QVLine (QFrame)	Vertical separation line.
ClickableLabel (QFrame)	Clickable frame, which shows an icon + text. It has a state indicating if the associated content is expanded or collapsed. Depending on the state, the icon changes to represent the current state. If the frame is clicked, a signal is emitted.
CustomComboBox (QComboBox)	Like a normal QComboBox with the difference that it additionally emits a signal if the popup button is clicked.
CustomButton (QPushButton)	Like a QPushButton with a different constructor having parameters for setting the text, icon resource path, and icon size.
DigitLineEdit (QLineEdit)	QLineEdit that only allows digits (other characters are ignored) and emits a signal if the number changes.
TaskButton (QPushButton)	Implements the view of a task button, which visualizes a text and an icon indicating the task type. Moreover, it provides a menu for deleting, activating, and deactivating a task. The menu is shown on a right-mouse click. Furthermore, the border color changes depending on the task state (activated - green, deactivated - red).

Table 4.1.: Custom widgets.

```python
import qdarkstyle
...
dark_stylesheet = qdarkstyle.load_stylesheet_pyqt5()
widget.setStyleSheet(dark_stylesheet)
...
);
```

Listing 4.1: Code for setting the style sheet of a widget (and nested ones) to the dark mode style sheet provided by the library QDarkStyle.

4.2. Data Storage

One design decision made in the previous chapter is to use SQLite for structuring our collected data. To understand the later implementation steps in this chapter it is essential to be familiar with the data tables and their relationships. Furthermore, the custom analysis feature of our tool is based on SQL Select statements, where the schema has to be known as well. Thus, the next part introduces the SQL schema with its tables and data relationships.

4.2.1. Datastructure & SQL Schema

Figure 4.2 shows the SQL schema representing the data structure for the collected data from Steam. It consists of several tables, where each has an attribute *id* for the primary key. Each attribute of a table has a type indicated by a single symbol or letter. The three data types, which occur at least once in the schema, are d (date and time), # (integer), and t (text). As can be seen, a project can have several applications, and an application can have several associated tags, genres, and game area details. As the same genre, tag, or game area detail can be part of more than one application, the intermediate tables *game_details_to_applications*, *genres_to_applications*, and *tags_to_applications* are used to create an N:N relationship between the *applications* table and the individual tables *genres*, *tags*, and *game_details*. The table *review_containers* contains fields for storing the summary of the collected reviews and links to the *applications* table. Each review record links

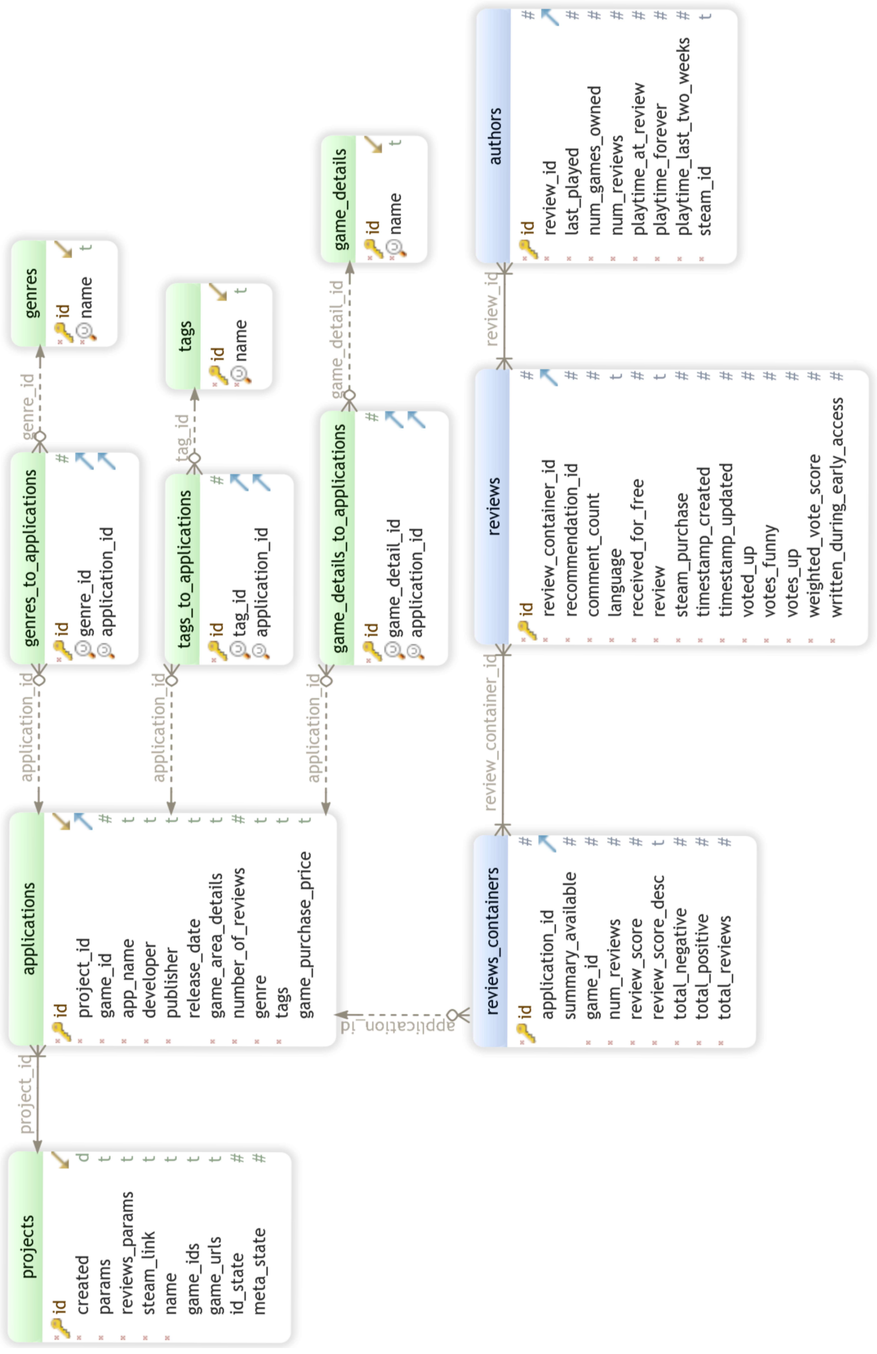


Figure 4.2.: SQL schema.

to one review container record. The same holds true for the relationship between the *reviews* table and the *authors* table. As a review usually has one author, and the tool is designed to allow one review container per application record, the corresponding 1:N relationships might be replaced by 1:1 relationships in a later release.

4.2.2. Building Python with SQLite Math Support

Although the SQLite3 library is directly shipped with Python, the built-in mathematical functions[4] are disabled by default. The math functions allow to use functions like $pow(X)$, $log(X)$, $exp(X)$, $sin(X)$, $sqrt(X)$ directly in a SQL statement and are advantageous for writing complex statements. To enable them, Python needs to be built with the compile option DSQLITE_ENABLE_MATH_FUNCTIONS. To achieve this, several steps are necessary:

1. Download Python zipped source code from Python releases site[5]
2. Unpack source code for Python in a folder of choice
3. Open PYTHON_FOLDER/PCbuild/sqlite3.vcxproj in a text editor and add the compile option DSQLITE_ENABLE_MATH_FUNCTIONS to the pre-processor definitions
4. Install Visual Studio (e.g. Visual Studio Community 2019)
5. Open PYTHON_FOLDER/PCbuild/pcbuild.sln in Visual Studio
6. Build the project

After the above steps have been performed, the build can be found in PYTHON_FOLDER/PCbuild/amd64 (Stinner, 2017). Another extension for SQLite, we want to utilize, is the FTS5 extension, which adds full-text search capabilities to a database. However, the FTS5 extension can not just be simply performed on any text attribute of a table. Instead, it must be defined beforehand, which attributes should be supported.

[4]https://www.sqlite.org/lang_mathfunc.html
[5]https://www.python.org/downloads/windows/

4.2.3. Integrating FTS5 Extension for Text Based Search

In the case of our proposed tool, we want to add full-text search functionality to the review text, i.e., the *review* attribute. This was achieved by following the tutorial of Lam (2020) and Hipp et al. (2021). Listing 4.2 shows the virtual table, where the syntax *USING fts5()* defines that the table should use the FTS5 extension. The *content* attribute has to match the name of table that contains the *review* attribute, and the *content_rowid* is defined to match the *reviews.id* (the primary key of the *reviews* table).

```sql
CREATE VIRTUAL TABLE reviews_fts USING fts5(
    review_container_id UNINDEXED,
    recommendation_id UNINDEXED,
    comment_count UNINDEXED,
    language UNINDEXED,
    received_for_free UNINDEXED,
    review,
    steam_purchase UNINDEXED,
    timestamp_created UNINDEXED,
    timestamp_updated UNINDEXED,
    voted_up UNINDEXED,
    votes_funny UNINDEXED,
    votes_up UNINDEXED,
    weighted_vote_score UNINDEXED,
    written_during_early_access UNINDEXED,
    content=reviews,
    content_rowid=id
);
```

Listing 4.2: Virtual table for adding full-text search capabilities to the review text.

Executing the statements creates a set of tables that enables indexing on the *reviews* table. As we only want the indexing to be applied on the *review* attribute, we define all other attributes to be *UNINDEXED*. SQL Select statements can be performed on the FTS5 table like on any other table, with the difference that it provides additional functionalities on the indexed attributes. An example can be seen in Listing 4.3. The statement returns a sorted list of reviews in which the term "cybersickness" occurs at least once. The sorting depends on the rank, which is based on the BM25 algorithm and indicates how well a row matches the query. However, the FTS5 extension

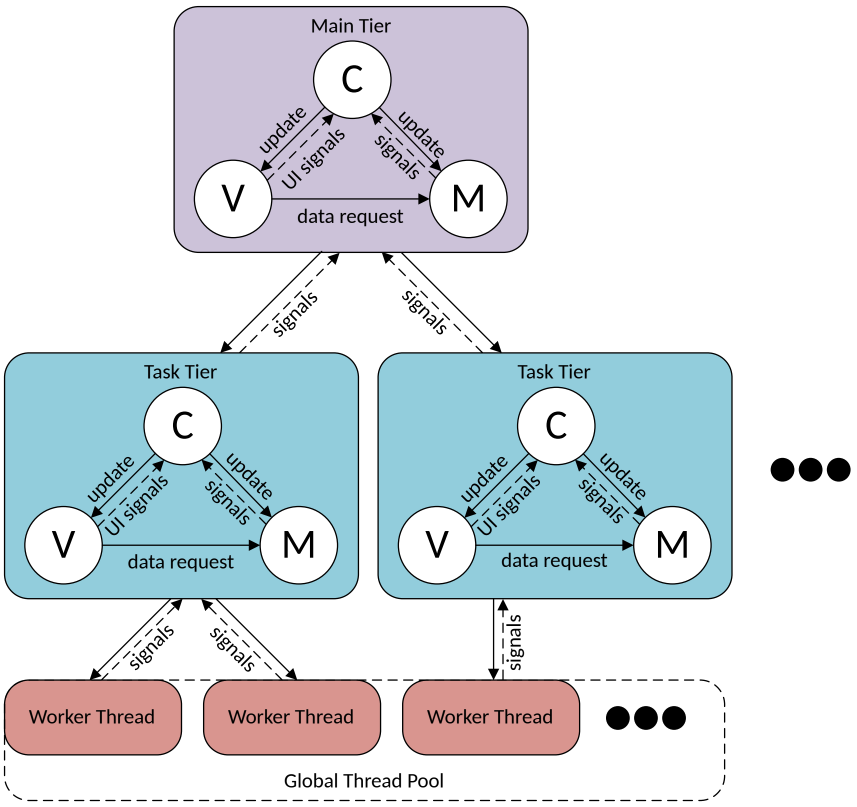


Figure 4.3.: Architecture for collection and analysis processes.

has a lot more functionalities, which are described on the SQLite website[6] of the SQLite developers Hipp et al. (2021).

```sql
SELECT * FROM reviews_fts WHERE review MATCH 'cybersickness'
    ORDER BY rank;
);
```

Listing 4.3: Example SQL Select statement using FTS5 functionalities.

4.3. Architecture

Although the processes for collecting data and the processes for analyzing the data differ in the tasks they perform, the fundamental architecture shown in Figure 4.3 is the same. It uses the HMVC and worker thread pattern described in a previous section. When a task is executed, a new MVC component is created. This component does not know anything about its parent. The parent model on the other hand stores a reference to the controller of its child and can access the child's public methods and variables. To start the actual execution of a task, the parent model calls the corresponding method of its child. As a result, the model of the child creates a queue of jobs and creates one or more worker threads that are submitted to the global thread pool. Each of the worker threads is given a unique ID for identification and a dictionary is used to map the ID to the worker thread. In a loop, a worker pops a job from the shared queue and executes it. This is done until the queue is empty.

4.3.1. Signals

To propagate information and states back in the architecture, signals are used. The model of the MVC component (main tier), which is responsible for creating tasks, connects to the signals of its children's controllers (task tier). Similarly, the MVC components which create worker threads connect to the signals of the workers. Furthermore, as the business logic is always handled by models and not by the controllers, the models emit signals to notify their controllers of changes. The controller processes the changes and triggers an update of the view (UI). Likewise, the controller also connects to the signals of the view and updates the model accordingly on a detected signal.

[6]https://www2.sqlite.org/fts5.html

4.4. Data Collection

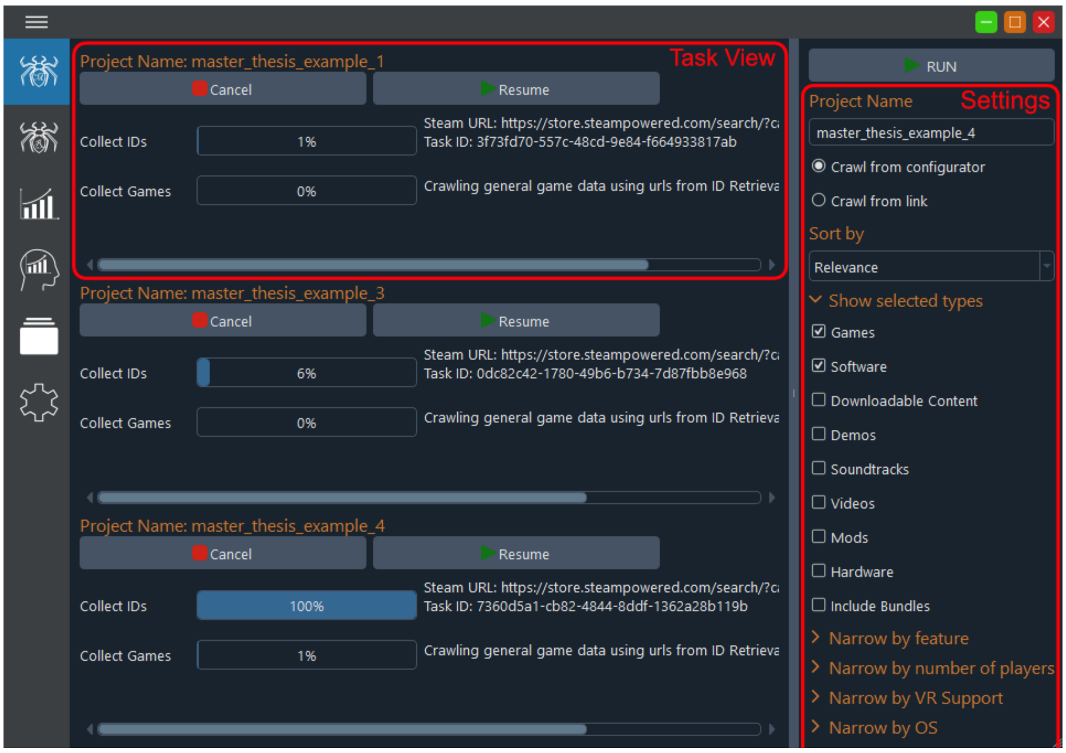
The data collection is split into the process for collecting general game data and the process for collecting reviews, where each of the two processes has a separate view. As the ID of a game is required to collect its reviews, this section starts by describing how the game IDs and the corresponding game data are gathered from Steam.

4.4.1. Collecting Game IDs and Game Data

As described in the previous chapter, a project always starts by collecting the game IDs and game data narrowed to games with specific characteristics concerning the type, feature, number of players, VR support, and OS support.

View

Figure 4.4 shows the view for narrowing games and starting a collection process. The view is shown when the application is started and takes the top position in the menu. One part corresponds to the settings defining which games should be collected, and the other part shows the progress of executed collection tasks. Each collection task has a separate view with control buttons for pausing, canceling, resuming, and closing. The task view has two progress bars, where the first one corresponds to the progress for collecting game IDs, and the other corresponds to the progress for collecting the game data associated with the game IDs. To visualize simultaneously executed tasks, task views are arranged in a vertical layout. This vertical layout is encapsulated into a scroll area with vertical scrolling enabled, which means that a scroll bar appears in the case the area for visualizing all tasks views at once is too small. Concerning the settings, a radiobutton group, consisting of two radiobuttons, specifies if the in-tool configurator or a Steam search URL should be used to define which games to collect. In the case of the in-tool configurator, the narrowing categories and their checkboxes are created dynamically by extracting the fields from the Steam



Project Name: master_thesis_example_1

Collect IDs 1% Steam URL: https://store.steampowered.com/search/?c
Task ID: 3f73fd70-557c-48cd-9e84-f664933817ab

Collect Games 0% Crawling general game data using urls from ID Retrieva

Project Name: master_thesis_example_3

Collect IDs 6% Steam URL: https://store.steampowered.com/search/?c
Task ID: 0dc82c42-1780-49b6-b734-7d87fbb8e968

Collect Games 0% Crawling general game data using urls from ID Retrieva

Project Name: master_thesis_example_4

Collect IDs 100% Steam URL: https://store.steampowered.com/search/?c
Task ID: 7360d5a1-cb82-4844-8ddf-1362a28b119b

Collect Games 1% Crawling general game data using urls from ID Retrieva

Project Name: master_thesis_example_4

[✓ Crawl from configurator] Crawl from link

Sort by: Relevance

Show selected types
- ☑ Games
- ☑ Software
- ☐ Downloadable Content
- ☐ Demos
- ☐ Soundtracks
- ☐ Videos
- ☐ Mods
- ☐ Hardware
- ☐ Include Bundles

Narrow by feature
Narrow by number of players
Narrow by VR Support
Narrow by OS

Figure 4.4.: View for collecting general game data.

search page. If a collection process is started, the user selection in the in-tool configurator is converted to a Steam search URL for further processing.

In-Tool Configurator

To automatically create the narrowing categories with their checkboxes in the view and gather the parameters for building the Steam search URL, the Steam search page[7] is requested, and the required data is extracted from the HTML via Scrapy's XPath selectors[8]. For visualizing the categories and their checkboxes, the category names and parameter names (checkbox names) are required. To further build the Steam URL from the selection,

[7]https://store.steampowered.com/search/?term=
[8]https://docs.scrapy.org/en/latest/topics/selectors.html

the parameter values and the category collapse names are needed, where a parameter value is associated with the parameter name and a data collapse name is associated with a category name. Furthermore, the parameters are part of only one category. The only values that are known beforehand are the data collapse names, which are manually extracted from the HTML. Listing 4.4 shows the Python code for extracting the category name and the associated parameters via XPath selectors for the data collapse name "vrsupport". In total, the data for the data collapse names "category1", "category2", "category3", "vrsupport", and "os", are extracted.

```python
from scrapy import Selector
...
selector = Selector(text=response.text)
category_name = selector.xpath('//div[@data-collapse-name="
    vrsupport"]/div[@class="block_header"]//text()').getall()
parameter_names = selector.xpath('//div[@data-param="vrsupport
    "]/@data-loc').getall()
parameter_values = selector.xpath('//div[@data-param="vrsupport
    "]/@data-value').getall()

print(category_name)
print(parameter_names)
print(parameter_values)

# Output
# ['Narrow by VR Support']
# ['VR Only', 'VR Supported', ..., 'Standing', 'Room-Scale']
# ['401', '402', ..., '302', '303']
...
```

Listing 4.4: Extracting category name and its associated parameters from the Steam search page.

Steam Search URL

To build the Steam search URL from the in-tool configurator selection, it is necessary to understand how the URL is built. Let us assume that the checkboxes "HTC Vive" and "Oculus Rift" from the category "Narrow by VR Support" and the checkbox "Windows" from the category "Narrow by OS" are ticked. In this case the Steam search URL visualized in Figure 4.5

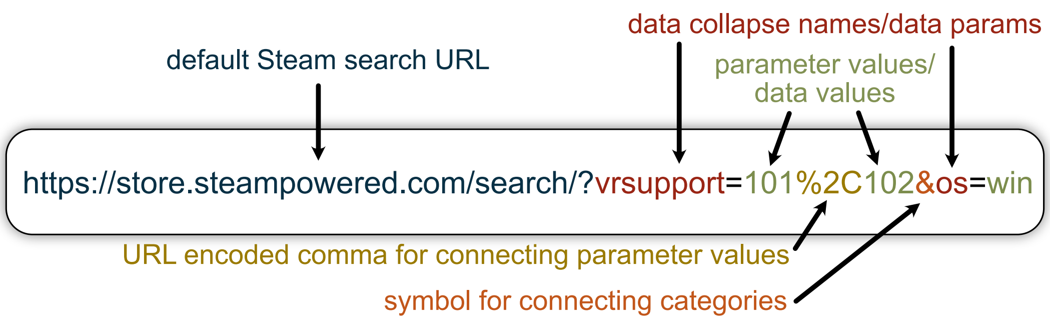

data collapse names/data params
default Steam search URL
parameter values/ data values
https://store.steampowered.com/search/?vrsupport=101%2C102&os=win
URL encoded comma for connecting parameter values
symbol for connecting categories

Figure 4.5.: Composition of the Steam search URL.

correctly reflects the described selection. As can be seen, parameter values from the same category are connected via %2C, which is the URL encoded representation of a comma. On the other hand, if checkboxes from more than one category are ticked, the categories are connected via the & symbol. Concerning the URL, a data collapse name corresponds to a URL parameter name, and a string resulting from a comma concatenation of the parameter values of the same category form a URL parameter.

Collecting Game IDs

The process for collecting the game IDs uses the Steam search URL, which is created in the case of the "Crawl from configurator" option or is directly provided in the case of the "Crawl from link" option. The whole procedure follows the previously described architecture. If the collection task is executed, a separate MVC component is created, which processes the task by utilizing a worker thread. However, to collect game IDs only one worker thread per collection task is used. The task description the worker obtains only consists of the Steam search URL. However, the Steam search URL is not used directly to request the associated HTML. The problem is that the Steam search page[9] implements infinite scrolling to load results, which means that at the start only a subset of the available results are visible and on scrolling down with the mouse, additional results are loaded. Instead of simulating a scrolling-down event, the network analyzer

[9]https://store.steampowered.com/search/?term=

of Firefox was utilized to find out which requests are made in the background to load the additional results. As a result, the Steam search URL has to be modified by replacing *https://store.steampowered.com/search/?* with *https://store.steampowered.com/search/results/?query&* and by adding additional parameters called *start*, *count*, *dynamic_data* and *infinite*. The *start* parameter defines the offset of results that should be loaded and the *count* parameter defines the number of results that should be in the response. The modified URL from Figure 4.5 with an offset (*start*) of 0 and a result count of 100 equals *https://store.steampowered.com/search/results/?query&start=0&count=100&dynamic_data=&sort_by=Name_ASC&vrsupport=401&snr=1_7_7_230_7&infinite=1*. The response of a performed HTTP request returns a JSON encoded file with the four objects, namely *success*, *result_html*, *total_count*, and *start*. The *success* object value is 1 on success and 0 on failure, the *total_count* object value corresponds to the total number of results, the *start* object value equals the *start* parameter value of the request URL, and the *result_html* object value contains the data of the results in HTML format. In detail, the *result_html* contains the required game URLs that should be extracted. The described conversion of the Steam search URL is done by the worker. The first URL has an offset of 0 and the *count* is always set to 100. In a loop, the worker performs an HTTP request, parses the *result_html* object value by extracting the URLs using XPath selectors, filters the extracted URLs, extracts the game IDs from the URLs, and adds the IDs to the result list. After each iteration, the offset (*start*) value is increased by 100. However, if the previous offset exceeds the *total_count* object value, the loop is left as there are no more results. Concerning the URL filtering step, URLs that do not correspond to a game, are neglected. Precisely, only URLs with the prefix *https://store.steampowered.com/app/* are considered. However, URLs with e.g. the prefix of *https://store.steampowered.com/bundle/* are ignored as they refer to game bundles, which are not supported by our tool. When all IDs have been extracted, the worker signals its parent node (caller) that the task is finished. As a result, the parent has a list of the relevant game IDs. For collecting the game information associated with the game IDs, one or more worker threads, which request and parse the individual game pages, are created depending on the preferences.

Collecting Game Data

The worker threads that are responsible for collecting the general game information, pop game IDs from the queue and create the corresponding game URL of the format *https://store.steampowered.com/app/xxxxxx/*, where the xxxxxx is replaced with the game ID. Having a URL, a worker requests the corresponding HTML. In the case of games with mature content, one problem is that a redirect to an age-check page happens. To avoid this behavior, a cookie with the name *birthtime* and a UTC encoded birth data (9th of January, 1996) as value, is sent along with the request. The attributes shown in Figure 4.6 are requested via XPath or CSS selectors[10], where the names reflect the names of the value in the database table *applications*. The extracted values are stored in an object, which is added to the result list that is shared between the workers and their common parent. If a worker thread has finished a job, it tries to pop a job from the queue again. If the queue is empty, a signal is emitted to notify the parent that the thread has finished. If all worker threads have finished, the result list is complete and consequently, all entries are stored in the database.

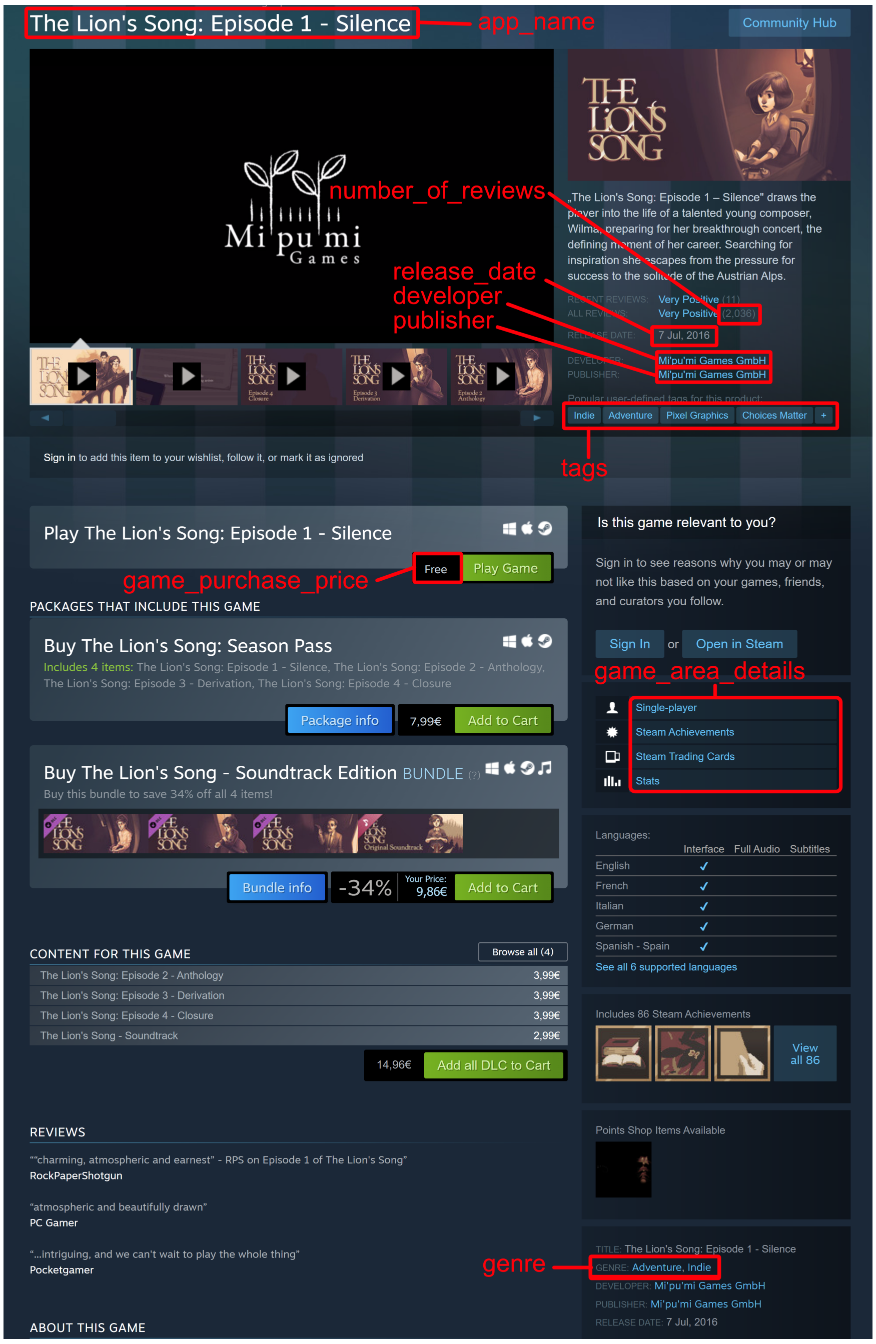
4.4.2. Collecting Reviews

After the IDs and game data associated with a project have been collected, the user can collect the corresponding reviews of the games.

View

The view visualized in Figure 4.7 looks similar to the view for collecting the IDs and general game data. The main difference concerns the settings frame. It shows a combo box that allows selecting the project for which the reviews should be collected. If the checkbox "Enable Crawl from" is checked, a date-edit widget appears for setting the start date. Reviews that were written before the entered date are neglected and are not collected if the review collection task is executed. However, this option only exists if the

[10]https://docs.scrapy.org/en/latest/topics/selectors.html



The Lion's Song: Episode 1 - Silence

Community Hub

„The Lion's Song: Episode 1 – Silence" draws the player into the life of a talented young composer, Wilma, preparing for her breakthrough concert, the defining moment of her career. Searching for inspiration she escapes from the pressure for success to the solitude of the Austrian Alps.

RECENT REVIEWS: Very Positive (11)
ALL REVIEWS: Very Positive (2,036)

RELEASE DATE: 7 Jul, 2016

DEVELOPER: Mi'pu'mi Games GmbH
PUBLISHER: Mi'pu'mi Games GmbH

Popular user-defined tags for this product:
Indie | Adventure | Pixel Graphics | Choices Matter | +

Sign in to add this item to your wishlist, follow it, or mark it as ignored

Play The Lion's Song: Episode 1 - Silence

Free | Play Game

PACKAGES THAT INCLUDE THIS GAME

Buy The Lion's Song: Season Pass

Includes 4 items: The Lion's Song: Episode 1 - Silence, The Lion's Song: Episode 2 - Anthology, The Lion's Song: Episode 3 - Derivation, The Lion's Song: Episode 4 - Closure

Package info | 7,99€ | Add to Cart

Buy The Lion's Song - Soundtrack Edition BUNDLE (?)

Buy this bundle to save 34% off all 4 items!

Bundle info | -34% | Your Price: 9,86€ | Add to Cart

CONTENT FOR THIS GAME | Browse all (4)

The Lion's Song: Episode 2 - Anthology	3,99€
The Lion's Song: Episode 3 - Derivation	3,99€
The Lion's Song: Episode 4 - Closure	3,99€
The Lion's Song - Soundtrack	2,99€

14,96€ | Add all DLC to Cart

REVIEWS

""charming, atmospheric and earnest" - RPS on Episode 1 of The Lion's Song"
RockPaperShotgun

"atmospheric and beautifully drawn"
PC Gamer

"...intriguing, and we can't wait to play the whole thing"
Pocketgamer

ABOUT THIS GAME

Is this game relevant to you?

Sign in to see reasons why you may or may not like this based on your games, friends, and curators you follow.

Sign In or Open in Steam

Single-player
Steam Achievements
Steam Trading Cards
Stats

Languages:

	Interface	Full Audio	Subtitles
English	✓		
French	✓		
Italian	✓		
German	✓		
Spanish - Spain	✓		

See all 6 supported languages

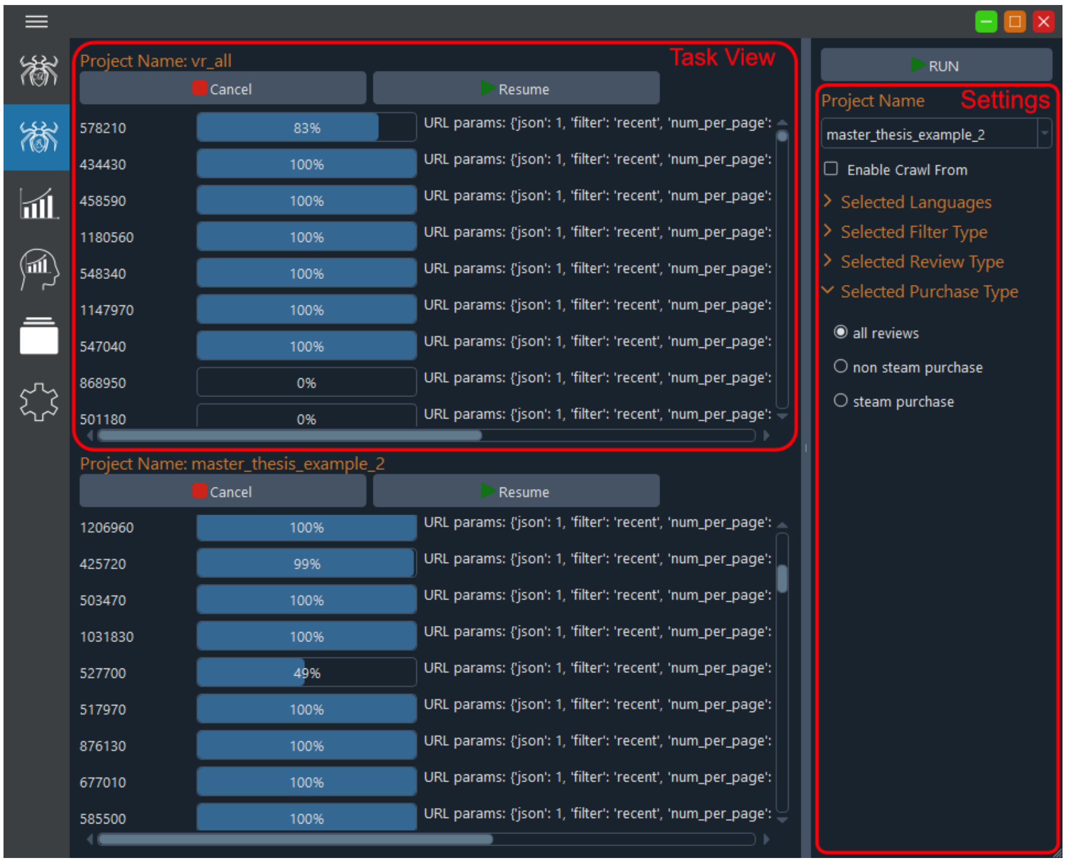
Includes 86 Steam Achievements

View all 86

Points Shop Items Available

TITLE: The Lion's Song: Episode 1 - Silence
GENRE: Adventure, Indie
DEVELOPER: Mi'pu'mi Games GmbH
PUBLISHER: Mi'pu'mi Games GmbH
RELEASE DATE: 7 Jul, 2016

Figure 4.6.: Steam game website screenshot of the game "The Lion's song: Episode 1 - Silence" of Mi'pu'mi Games GmbH with highlighted extracted information.



Project Name: vr_all

578210	83%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
434430	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
458590	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
1180560	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
548340	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
1147970	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
547040	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
868950	0%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
501180	0%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':

Project Name: master_thesis_example_2

1206960	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
425720	99%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
503470	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
1031830	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
527700	49%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
517970	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
876130	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
677010	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':
585500	100%	URL params: {'json': 1, 'filter': 'recent', 'num_per_page':

Project Name: master_thesis_example_2

☐ Enable Crawl From

[✓ all reviews] non steam purchase steam purchase

Figure 4.7.: View for collecting reviews.

reviews are sorted by helpfulness, i.e., the filter type "Review helpfulness" is used. Moreover, the right sidebar allows narrowing the reviews to specific languages and a purchase type. The latter one defines if the author who wrote the review has paid for the game on Steam. When a task is executed, a task view is added to the content frame. The task view has control buttons to control the execution of the task, i.e., controls to pause, cancel, resume, or close a task. Each game has a separate progress bar indicating what percentage of reviews of a game have already been collected.

Parameter	**Description**
filter	The parameter defines the sorting of the results. The allowed values are *recent* (sort by creation time), *updated* (sort by updated time), and *all* (sort by helpfulness).
language	The parameter allows narrowing the reviews to certain languages. The parameter value is a comma-separated string of API language codes [11] that should be considered. The parameter value *all* defines that all reviews should be responded.
day_range	The parameter defines how many days to look into the past. This parameter is only applicable for the *filter* parameter with the value *all*.
cursor	The results of a GET request are returned in chunks of 20 to 100 reviews depending on the *num_per_pages* parameter. The value of the start cursor is *. Each response contains the cursor for the next chunk. To correctly use the cursor, its value must be URL encoded.
review_type	The parameter defines which types of reviews should be collected. The allowed values are *positive*, *negative*, and *all*. In the case of our application, currently only the value *all* is supported.
purchase_type	The parameter defines if all reviews (*all* - default), only reviews written by individuals who did not pay for the game (*non_steam_purchase*), or only reviews written by individuals who paid for game (*steam*), are considered.
num_per_pages	The parameter defines how many reviews should be returned per GET request. The default value is 20 and the maximum allowed value is 100.

Table 4.2.: URL parameter description for queuing reviews via the Steam Web API (modified from Valve (2021e)).

Steam API - Get User Reviews

For collecting the reviews, a public method of the Steam Web API[12] is utilized, where the reviews can be collected via HTTP GET requests. The basic URL has the format *store.steampowered.com/appreviews/xxxxx?json=1*, where the xxxxx has to be replaced with the game ID. The parameter *json* with the value 1 defines that the result should be JSON encoded. A list of further parameters, with a description and allowed parameter values, is provided in Table 4.2. As can be observed, there is a strong connection between the right sidebar of the view and the URL parameters. However, the checkbox or radiobutton names do not directly correspond to the parameter values. Instead, more understandable names are used to make it easier for the user to understand their meaning. In the case of the language checkboxes, the API language codes are replaced with the complete language name. To map the checkboxes to the corresponding parameter values, a dictionary that maps the names of the checkboxes to the parameter values is created. When the review collection task is executed, a dictionary with the parameter name as key and the parameter value as value, is generated and passed to the MVC component that reflects the new task. Similarly, to the process for collecting game IDs and general game data, the game IDs that are connected with the selected project are pushed onto a job queue, and worker threads are utilized to collect the reviews. The worker threads pop a game id from the queue and create the request URL taking the URL parameters from the user selection into account. The URL parameter *num_per_page* is set to the maximum (100) for each request. The parameter *cursor* of the first request for collecting the reviews of a game is set to *. Along with the review results, the response delivers a query summary with information about the review score (rating of the game), and the number of reviews (positive, negative, total). The results are parsed and each review result is stored in a class object and added to the local result list. As mentioned in Table 4.2, the *cursor* to the next results chunk is part of the current response. As a result, the *cursor* parameter value of the request URL is modified and the next results are requested via an HTTP GET request. This is done until no more results are available. This might be the case if the response contains zero results. However, it turned out that the response sometimes contains

[12] https://partner.steamgames.com/doc/store/getreviews

zero reviews, although there should be more reviews available. Performing the same request again solves the problem. Therefore, an empty review list in the response is insufficient as a termination condition. However, the *total_reviews* value of the query summary allows the definition of a valid termination condition. Consequently, it is checked if the number of collected reviews matches the total number of reviews. If this is the case, the result list containing the collected reviews of the currently handled game ID as well as the corresponding review summary are stored in the database. As a result, the worker thread is free to execute the next job from the queue.

Having game data and (optionally) reviews stored in the SQLite database, the user can analyze the data. However, before going into detail, how the general or a custom analysis is performed in our tool, it is essential to understand how data is visualized.

4.5. Data Visualization

As previously mentioned, each analysis task is associated with a chart or table. To keep the business logic separated from the presentation, a model and view are created for each analysis task. When a task is executed, the model is updated, and afterward, an update of the view is triggered. At the current state, our tool supports bar charts, grouped bar charts, line charts, and overlapping histograms.

4.5.1. Chart Models

As can be seen in the UML class diagram (Figure 4.8), the chart models have a common base class from which they inherit variables and methods. The base class *ChartModel* is an abstract class with two abstract methods that have to be overwritten. The abstract method *update* is public and has the purpose to update the model based on the current values of the variables *sql_statement*, *preprocessing_function*, *python_preprocessing_enabled* and *python_preprocessing_code*. The *sql_statement* stores the SQL Select statement as a string, where the result of a request must fulfill certain characteristics

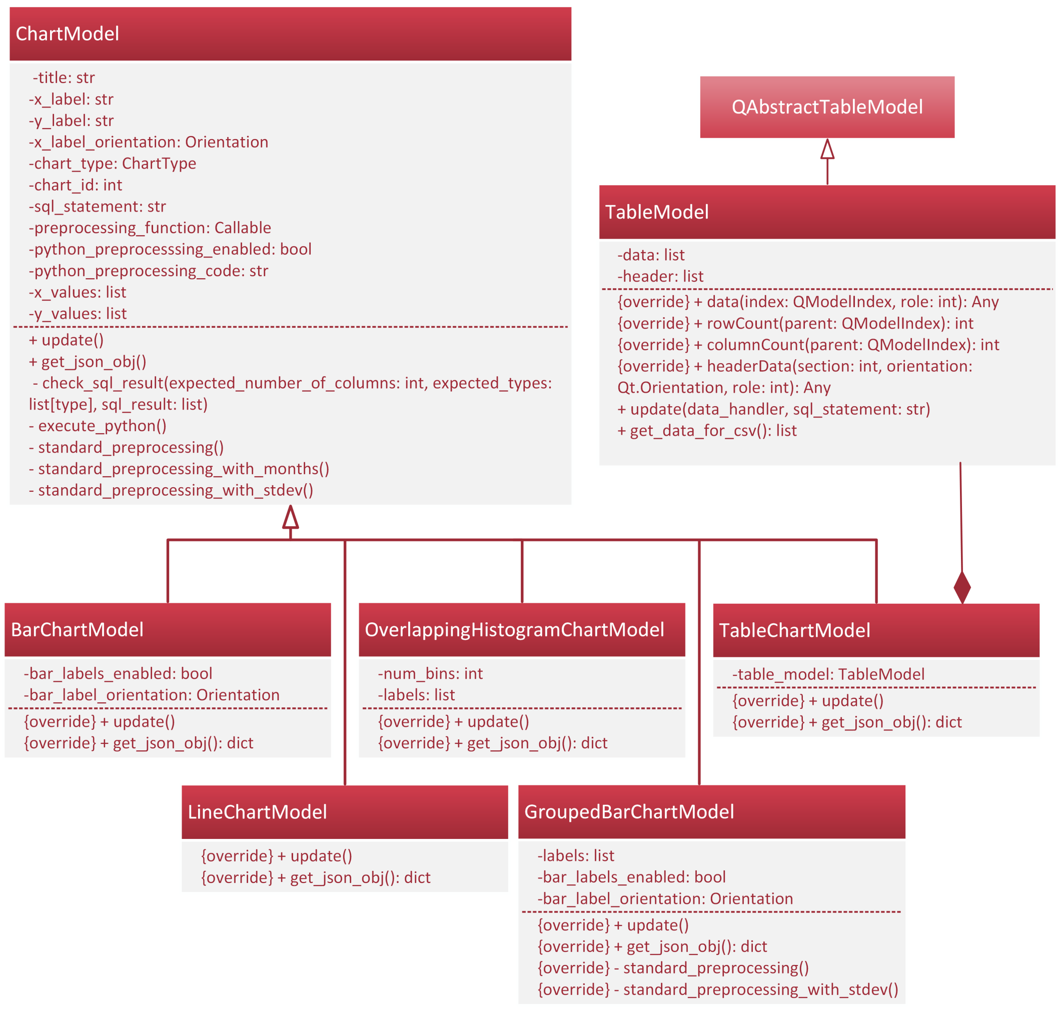

ChartModel
-title: str
-x_label: str
-y_label: str
-x_label_orientation: Orientation
-chart_type: ChartType
-chart_id: int
-sql_statement: str
-preprocessing_function: Callable
-python_preprocesssing_enabled: bool
-python_preprocessing_code: str
-x_values: list
-y_values: list
+ update()
+ get_json_obj()
- check_sql_result(expected_number_of_columns: int, expected_types: list[type], sql_result: list)
- execute_python()
- standard_preprocessing()
- standard_preprocessing_with_months()
- standard_preprocessing_with_stdev()
QAbstractTableModel
TableModel
-data: list
-header: list
{override} + data(index: QModelIndex, role: int): Any
{override} + rowCount(parent: QModelIndex): int
{override} + columnCount(parent: QModelIndex): int
{override} + headerData(section: int, orientation: Qt.Orientation, role: int): Any
+ update(data_handler, sql_statement: str)
+ get_data_for_csv(): list
BarChartModel
-bar_labels_enabled: bool
-bar_label_orientation: Orientation
{override} + update()
{override} + get_json_obj(): dict
OverlappingHistogramChartModel
-num_bins: int
-labels: list
{override} + update()
{override} + get_json_obj(): dict
TableChartModel
-table_model: TableModel
{override} + update()
{override} + get_json_obj(): dict
LineChartModel
{override} + update()
{override} + get_json_obj(): dict
GroupedBarChartModel
-labels: list
-bar_labels_enabled: bool
-bar_label_orientation: Orientation
{override} + update()
{override} + get_json_obj(): dict
{override} - standard_preprocessing()
{override} - standard_preprocessing_with_stdev()

Figure 4.8.: UML class diagram of chart models.

depending on the chart type and pre-processing function. The characteristics concern the column count and the column data types. To check if the SQL result is valid, the base class method *check_sql_result* is implemented. It takes the expected number of columns, a list of lists specifying the supported data types per column, and the SQL result as arguments. If the SQL result does not fulfill the characteristics, an error is thrown. If the *update* method is called, at first the SQL Select statement is executed and afterward the function *preprocessing_function* is called, which can be either *standard_preprocessing*, *standard_preprocessing_with_months* or *standard_preprocessing_with_stdev*. In the case of *standard_preprocessing* of the base class, the expected column count of the SQL result is two. The first column corresponds to the x-values and the second one to the y-values of a chart. In the function, the validity of the SQL result is checked, and the x-values and y-values are extracted. On the other hand, *standard_preprocessing_with_months* expects three columns. The first column corresponds to years, the second to months in form of integer values between one and twelve, and the third column to the y-values. The values of the first two columns are concatenated with a space delimiter and form the x-values. The *standard_preprocessing_with_stdev* function is similar to the *standard_preprocessing* function with the difference that it expects three columns, where the third column represents the standard deviation values and the second column represents mean values. The first column values are stored in the variable *x_values*, the second column values in variable *y_values* and the third column values are converted to variance values and are stored in the variable *variances*. In the case of the grouped bar chart model, the pre-processing functions are overwritten as an additional column is required to represent the label of a bar.

4.5.2. Table Model

However, the model-view implementation of the table differs from the other ones as it makes use of Qt's *QAbstractTableModel*[13] and *QTableView*[14] for visualizing tables. The *TableChartModel* has a class member called *table_model*, which is of the type *TableModel* and is automatic initialized when an object

[13]https://doc.qt.io/qtforpython-5/PySide2/QtCore/QAbstractTableModel.html
[14]https://doc.qt.io/qt-5/qtableview.html

is created. The *TableModel* is responsible for providing the data that will be displayed by the view. It is derived from *QAbstractTableModel* and must implement at least the methods *rowCount*, *columnCount* and *data*. The *data* method returns the data at a specific index (row, column) and the method *headerData* returns the header of a row or column at a specific index. The *TableModel* further implements a method *get_data_for_csv* which has the purpose to return a tabulator-separated CSV representation of the *data* in form of a string. In contrast to the other chart models classes, the table model class does not support pre-processing functions. Instead the *update* method handles the complete process for updating the corresponding model.

4.5.3. Dynamic Python Code

An advanced feature of our tool is that the SQL results can be modified via Python scripts. If the *python_preprocessing_enabled* variable has the boolean value *True*, the responsible method *execute_python* is called immediately after the SQL Select statement has been executed. The Python code stored in *python_preprocessing_code* is executed in a separate process. The reason for this is to keep the execution completely separate from the implementation of the tool itself. To communicate with the child process, a shared dictionary is created. This dictionary is used to pass the SQL result to the child process, which compiles and executes the Python code. To allow the Python code to access the SQL result, a mapping object is created in form of a dictionary. This dictionary is used to define which values should be accessible by the executed code and is passed as an argument to the *exec* function. A code excerpt of the implementation is shown in Listing 4.5. The *Manager* class as well as the *Process* class are part of the *multiprocessor*[15] library, a standard Python library.

```
@staticmethod
def execute_python_in_process(m_d, python_preprocessing_code):
  compiled_code = compile(python_preprocessing_code, '
   py_custom_script', 'exec')

  local_dict = dict()
  local_dict['values'] = m_d['values']
```

[15]https://docs.python.org/3/library/multiprocessing.html

```
  try:
    exec(compiled_code, globals(), local_dict)
  except Exception as e:
    m_d['err'] = e

  m_d['values'] = local_dict['values']

def execute_python(self):
  if not self.python_preprocessing_enabled:
    return

  manager = Manager()
  m_d = manager.dict()
  m_d['values'] = self.latest_sql_result

  process = Process(target=self.execute_python_in_process, args
    =(m_d, self.python_preprocessing_code))
  process.start()
  process.join()

  """ Error handling is not shown as part of this excerpt"""

  self.latest_sql_result = m_d['values']
```

Listing 4.5: Code excerpt for executing dynamic Python code in a separate process.

4.5.4. Chart Views

Each chart model class shown in Figure 4.8 has a corresponding view class in Figure 4.9. For the charts, the plotting library Matplotlib[16] is utilized, which provides a wide range of highly customizable plots. Moreover, the plots are easily integrable into a user interface created with PyQt5. As can be seen in Figure 4.9, the *ChartCanvas* is derived from *FigureCanvasQTAgg* which is a canvas object and behaves like any other Qt widget. In the constructor of the *ChartCanvas* a Matplotlib figure with axes is created, where the figure is passed to the constructor of its base class. The *ChartCanvas* is the base class of the individual chart canvases (views). Each derived class has a method *update_canvas_from_model* which resets the figure and draws the chart with

[16]https://matplotlib.org/

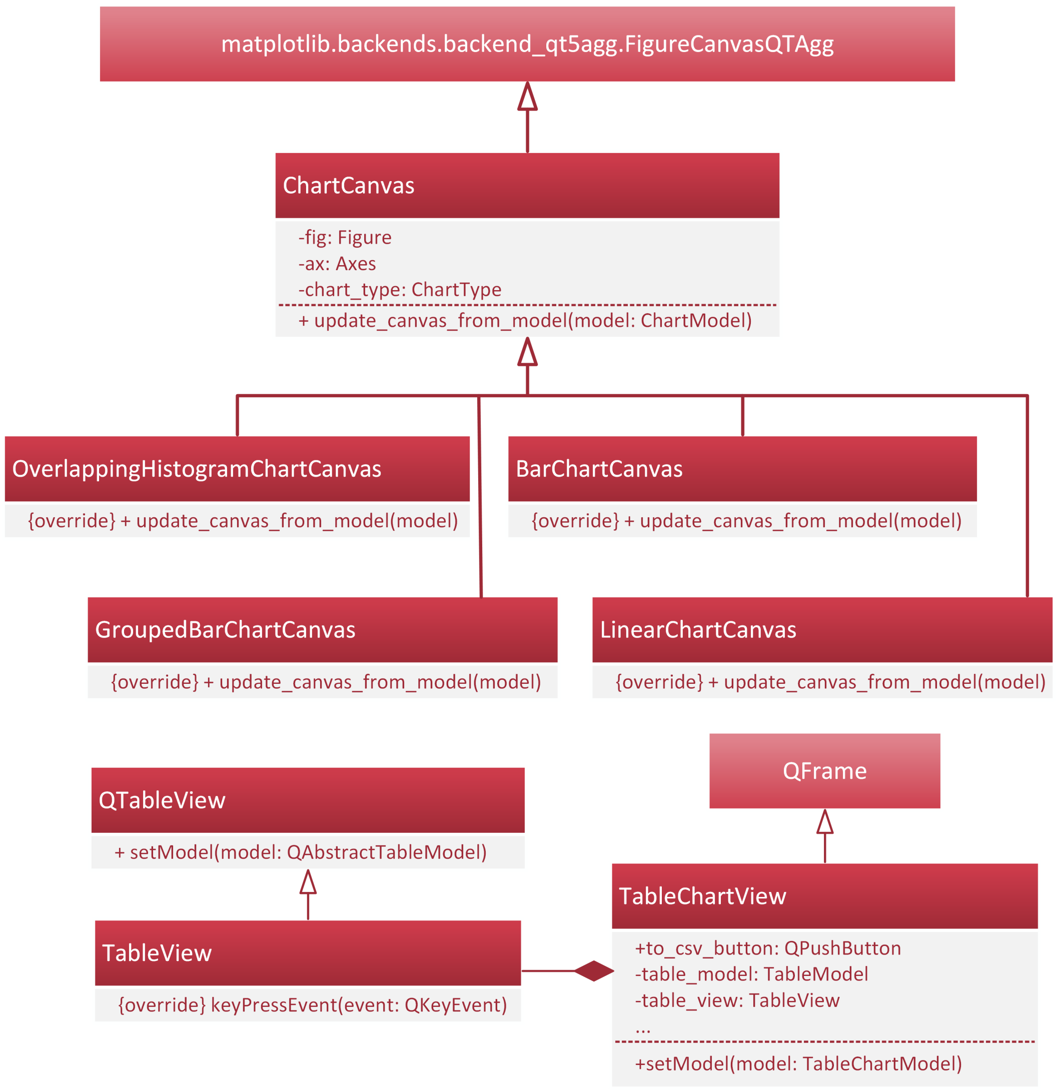
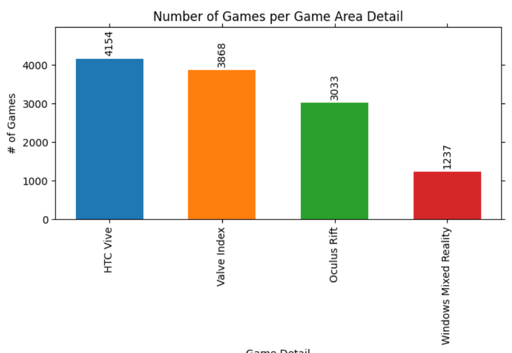
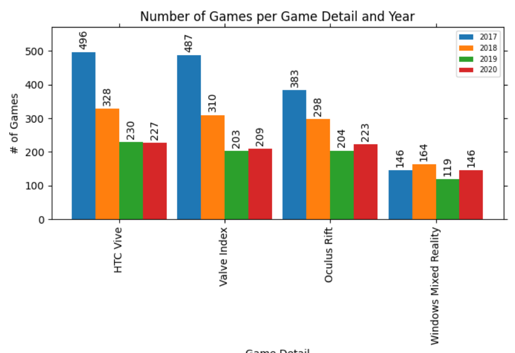

matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg
ChartCanvas
-fig: Figure
-ax: Axes
-chart_type: ChartType
+ update_canvas_from_model(model: ChartModel)
OverlappingHistogramChartCanvas
{override} + update_canvas_from_model(model)
BarChartCanvas
{override} + update_canvas_from_model(model)
GroupedBarChartCanvas
{override} + update_canvas_from_model(model)
LinearChartCanvas
{override} + update_canvas_from_model(model)
QTableView
+ setModel(model: QAbstractTableModel)
TableView
{override} keyPressEvent(event: QKeyEvent)
QFrame
TableChartView
+to_csv_button: QPushButton
-table_model: TableModel
-table_view: TableView
...
+setModel(model: TableChartModel)

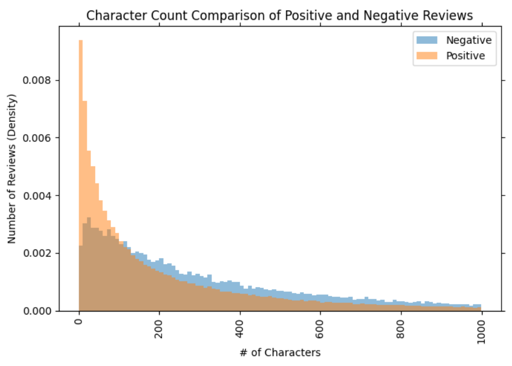
Figure 4.9.: UML class diagram of chart views.



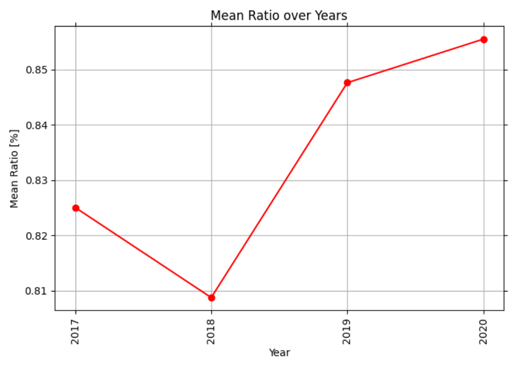
(a) Bar Chart



(b) Grouped Bar Chart



(c) Overlapping Histogram



(d) Line Chart

Figure 4.10.: Chart examples.

the respective chart model as input. One example for each chart type is visualized in Figure 4.10.

Matplotlib Navigation Toolbar

The Matplotlib library provides the possibility to add a navigation toolbar widget in PyQt5, which allows changing different properties of the chart concerning the axes labels, axes scaling, title, and label position. Furthermore, the widget has a button for zooming into the chart and a button for moving the content of the chart. Another button allows saving the chart to an image file. If the save button is clicked, a save file dialog is opened, which enables the user to set the name and path, where the image should be saved. To add the toolbar, *from matplotlib.backends.backend_qt5agg.NavigationToolbar2QT* must be imported. The constructor of *NavigationToolbar2QT* takes two parameters, a Matplotlib canvas and a parent widget like a *QFrame*.

4.5.5. Table View

The view for the table is built using Qt's *QTableView* widget, which has a spreadsheet-like look and capabilities (Fitzpatrick, 2020). The class provides a method for setting the model that contains the data. The model must be an object of a class that is derived from *QAbstractTableModel* like described in Subsection 4.5.2. The *TableView* class shown in Figure 4.9 is derived from *QTableView* to allow adding additional functionalities. *TableChartView* class has a variable *table_view* of type *TableView* which is created when an object is created. Furthermore, the class has a variable *table_model* of type *TableModel*, which can be set via the class method *setModel*. The method takes an *TableChartModel* as input that stores the required *TableModel* object. A reference to this object is stored in the variable *table_model*. The *TableChartView* itself is derived from Qt's frame widget *QFrame*. On initiate, the class sets the layout of the frame and adds the widgets that should be contained in the frame. These widgets are a button for converting the table content to a CSV file, and the *TableView* widget.

A *QTableView* widget allows selecting multiple cells like in other spreadsheet applications. To allow copying the selected content, the method *keyPressEvent* is overwritten. If the key event matches the copy shortcut of the operating system (Windows: Crtl + C), the selected cell indices are requested and a string is created, where the selected cells in the same row are tabulator-separated and the rows are newline-separated. This allows pasting the content directly into spreadsheet applications like Microsoft Excel.

4.6. General Data Analysis

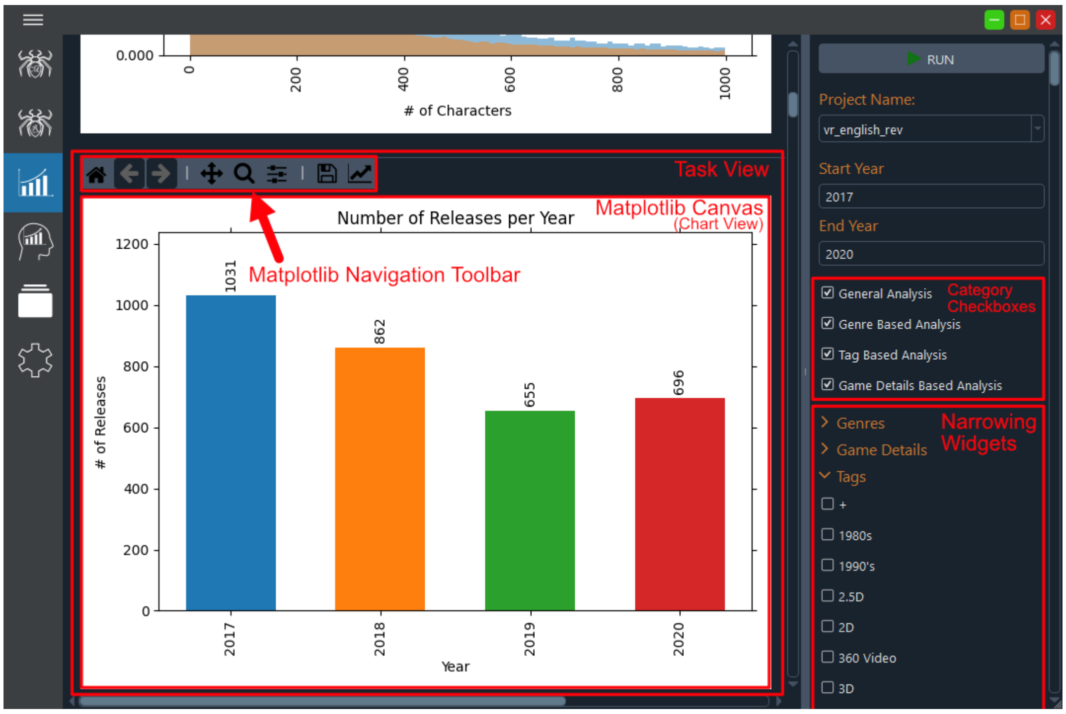
The previous section details how data can be visualized via charts or tables but does not describe the integration in the tool. The tool has a functionality for performing general analysis, where predefined charts are created automatically for the selected project.

4.6.1. View

The general analysis view visualized in Figure 4.11 corresponds to the third point in the menu sidebar. The settings frame, i.e., the right sidebar, has a combo box for selecting the project for which the general analysis tasks should be executed. Moreover, for time-based tasks, the output of the charts can be restricted to a time range defined by the two line-edit widgets below the project name combo box. As previously mentioned, each chart is associated with exactly one task. Each task is further assigned to one of the four categories *General Analysis*, *Genre-Based Analysis*, *Tag-Based Analysis*, or *Game-Detail-Based Analysis*. In the view, each category can be disabled or enabled via a separate checkbox. Tasks that are part of a disabled category are neither executed nor visualized. Except for tasks that are part of the *General Analysis* category, the output of the tasks can be restricted to specific genres, tags, and game area details via checkboxes. A Matplotlib chart resulting from an execution is encapsulated in a frame together with a navigation toolbar. All frames are arranged in a vertical layout.

4.6.2. Tasks

The properties of each task are predefined, which is done by creating a chart model per task at the time when the model associated with the general analysis is initialized. Each chart model is given a hard-coded ID for identification. A dictionary *_chart_id_model_dict*, which is a member of the general analysis model, maps the ID to the generated model. The Listing 4.6 shows the creation of the chart model of the chart visualized in Figure 4.11.



Project Name: vr_english_rev
Start Year: 2017
End Year: 2020

- ☑ General Analysis
- ☑ Genre Based Analysis
- ☑ Tag Based Analysis
- ☑ Game Details Based Analysis

- ☐ +
- ☐ 1980s
- ☐ 1990's
- ☐ 2.5D
- ☐ 2D
- ☐ 360 Video
- ☐ 3D

Figure 4.11.: View for general data analysis.

```
...
num_releases_per_year = BarChartModel(title='Number of Releases
    per Year', x_label='Year', y_label='# of Releases',
    chart_id=NUM_OF_RELEASES_PER_YEAR)

self._chart_id_chart_model_dict[NUM_OF_RELEASES_PER_YEAR] =
    num_releases_per_year
...
```

Listing 4.6: Creation of a predefined chart model.

To assign a chart to a category, another dictionary *CHART_GROUP_DICT* exists, where a key is a string representing the category and the value is a list containing the associated IDs of the assigned charts. To change the category membership, only the dictionary has to be manipulated. The Table 4.3 lists the current predefined charts that are available and gives information about their chart type and category. A description of the charts is not provided as the title already provides a good explanation.

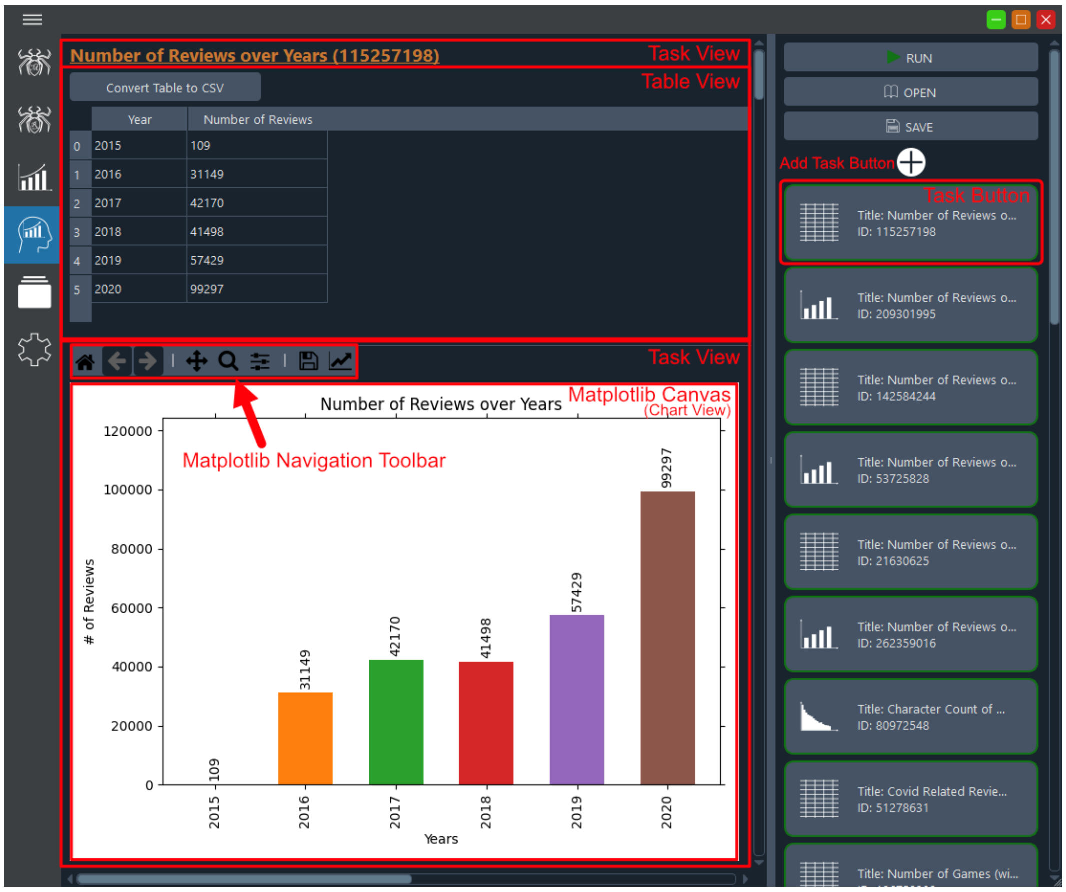
4.6.3. Execution

When the tasks are executed via a click on the *RUN* button, the chart models are updated. The update is done by a set of worker threads. The threads get the project name, a queue of jobs, and the current user preferences as input. The job queue is a simple list that contains the chart models that need to be updated. Which chart models are contained in the list depends on the checked categories. The created threads pop and execute a job from the queue in a loop until there are no more jobs left. The threads have a method called *__update_chart(chart_model: ChartModel)*. Depending on the currently handled chart model's ID, the method calls another method of the class *SQLCreator* that returns a SQL Select statement as a string. As input, the different methods require specific parameters like the project name, the set time range, or a list of selected genres, tags, or game area details depending on what is necessary to form the SQL Select statement to obtain the desired data. Having the SQL Select statement, the chart model is updated as mentioned in Section 4.5.1.

Title	**Type**	**Cat.**
Distribution of Steam Ratings	B	GA
Character Count Comparison of Positive and Negative Reviews	H	GA
Number of Releases per Year	B	GA
Number of Releases per Month	B	GA
Mean Ratio over Months	B	GA
Mean Ratio over Years	B	GA
Mean Ratio over Years	L	GA
Average Character Count over Years - All Reviews	B	GA
Average Character Count over Years - Positive Reviews	B	GA
Average Character Count over Years - Negative Reviews	B	GA
Average Character Count over Months - All Reviews	B	GA
Average Character Count over Months - Positive Reviews	B	GA
Average Character Count over Months - Negative Reviews	B	GA
Number of Games per Genre	B	GBA
Mean Ratio of Games per Genre	B	GBA
Number of Releases (Games) per Genre and Year	GB	GBA
Number of Games per Genre with Respect to Tags	GB	GBA
Number of Games per Genre with Respect to Game Area Details	GB	GBA
Number of Games per Tag	B	TBA
Mean Ratio of Games per Tag	B	TBA
Number of Releases (Games) per Tag and Year	GB	TBA
Number of Games per Tag with Respect to Genres	GB	TBA
Number of Games per Tag with Respect to Game Area Details	GB	TBA
Number of Games per Game Area Detail	B	GDBA
Mean Ratio of Games per Game Area Detail	B	GDBA
Number of Releases (Games) per Game Area Detail and Year	GB	GDBA
Number of Games per Game Area Detail with Respect to Genres	GB	GDBA
Number of Games per Game Area Detail with Respect to Tags	GB	GDBA

Table 4.3.: Description of general analysis charts. The type column defines if the chart is a bar chart (B), a grouped bar chart (GB), a line chart (L), or a histogram (H). The category column defines if the chart is part of the *General-Analysis* (GA), *Genre-Based Analysis* (GBA), *Tag-Based Analysis* (TBA) or *Game-Detail-Based Analysis* (GDBA)

.



	Year	Number of Reviews
0	2015	109
1	2016	31149
2	2017	42170
3	2018	41498
4	2019	57429
5	2020	99297

Title: Number of Reviews o...
ID: 115257198

Title: Number of Reviews o...
ID: 209301995

Title: Number of Reviews o...
ID: 142584244

Title: Number of Reviews o...
ID: 53725828

Title: Number of Reviews o...
ID: 21630625

Title: Number of Reviews o...
ID: 262359016

Title: Character Count of ...
ID: 80972548

Title: Covid Related Revie...
ID: 51278631

Title: Number of Games (wi...

Figure 4.12.: View for custom data analysis.

4.7. Custom Data Analysis

While the previous section describes the implementation to run an analysis that outputs a set of predefined charts, this section discusses the implementation that allows users to generate custom tasks via SQL Select statements, whose result can be visualized via charts or tables.

4.7.1. View

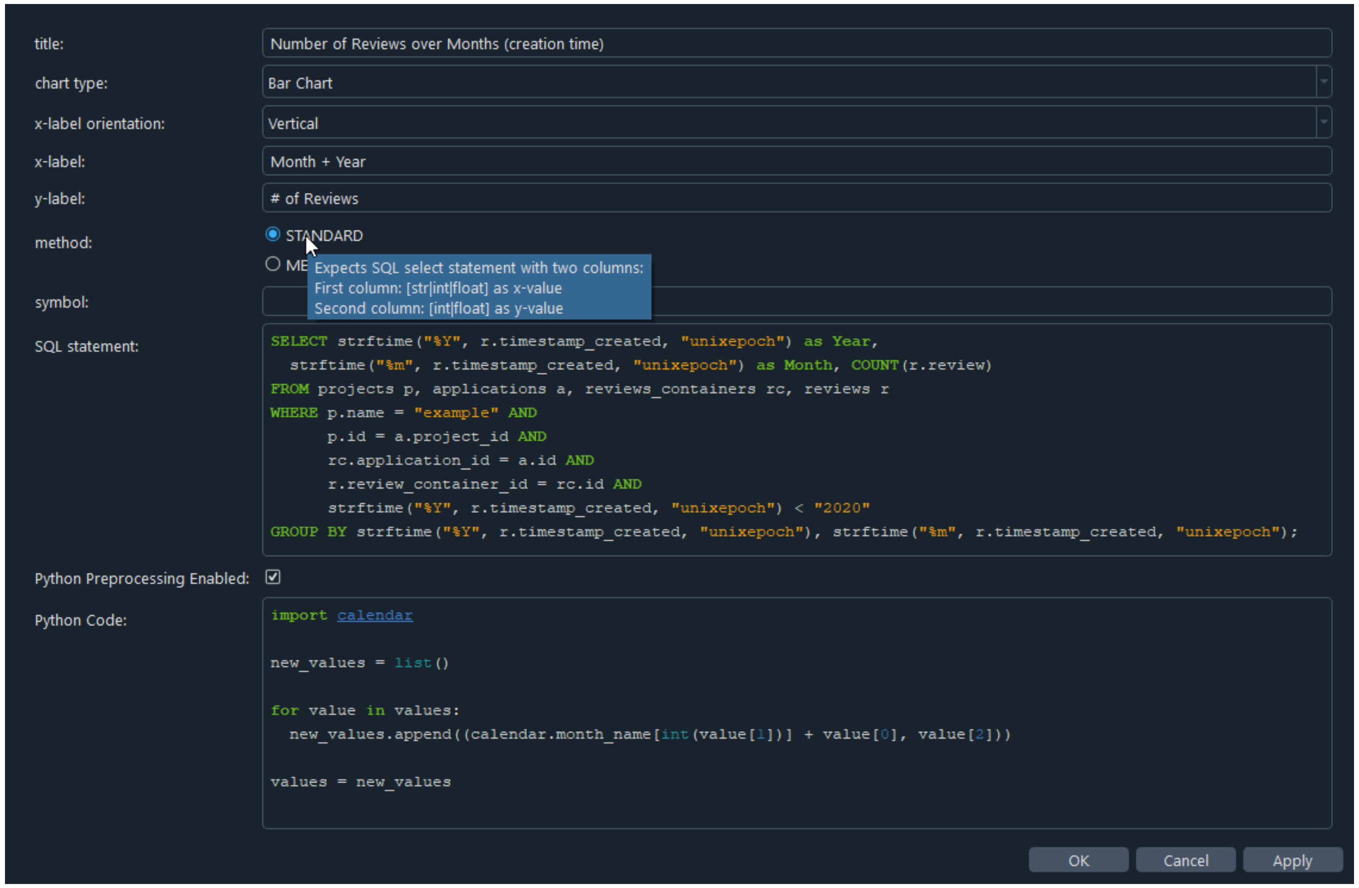
The view shown in Figure 4.12 is associated with the fourth point in the menu sidebar. The right sidebar has a button for running the custom tasks. Furthermore, it has buttons for saving and opening self-defined tasks. Moreover, it has a button to add a custom task. When the button is clicked, a separate dialog opens, which allows configuring the task. When a task is added, it is displayed in form of a custom button in the right sidebar. A task button encapsulates an image representing the type of task (e.g., bar chart, table, etc.), the task title, and the task ID. In addition, each task button has a context menu, which opens on a right-mouse click on the task button. The context menu has buttons for activating, deactivating, or deleting a task. The area that contains the charts and tables is similar to the one from Figure 4.11, and only differs in aspects that cannot be observed by users.

Task Dialog View

The dialog that opens on a mouse click on the button for adding a task, is visualized in Figure 4.13. It encapsulates a frame with a form layout and general dialog buttons. The form layout has different fields for configuring a task. Depending on the currently selected chart type, more or fewer fields are available. The fields are automatically updated when the chart type is changed. In general, the available fields reflect the configurable variables/properties of the respective chart model or table model in Figure 4.8.

4.7.2. Task Creation

When a task is created or updated via the dialog button *Apply*, a chart model of the selected subclass is initialized using the data from the dialog. Furthermore, each newly created chart model is given a unique ID. The model is added to a dictionary using its ID as the key.



title: Number of Reviews over Months (creation time)

chart type: Bar Chart

x-label orientation: Vertical

x-label: Month + Year

y-label: # of Reviews

method: [✓ STANDARD] ME

Expects SQL select statement with two columns:
First column: [str|int|float] as x-value
Second column: [int|float] as y-value

symbol:

SQL statement:

```
SELECT strftime("%Y", r.timestamp_created, "unixepoch") as Year,
  strftime("%m", r.timestamp_created, "unixepoch") as Month, COUNT(r.review)
FROM projects p, applications a, reviews_containers rc, reviews r
WHERE p.name = "example" AND
      p.id = a.project_id AND
      rc.application_id = a.id AND
      r.review_container_id = rc.id AND
      strftime("%Y", r.timestamp_created, "unixepoch") < "2020"
GROUP BY strftime("%Y", r.timestamp_created, "unixepoch"), strftime("%m", r.timestamp_created, "unixepoch");
```

Python Preprocessing Enabled: [✓]

Python Code:

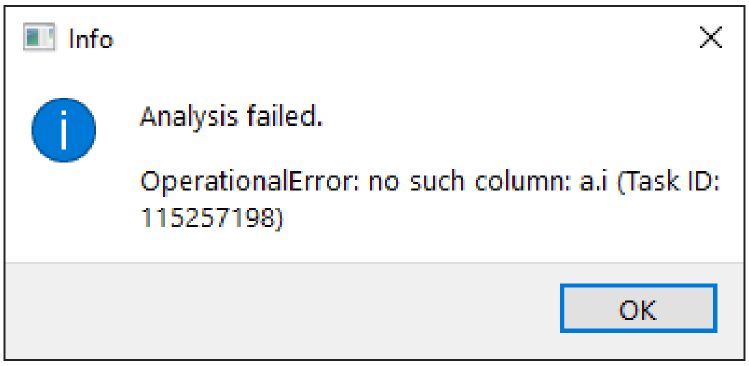
```
import calendar

new_values = list()

for value in values:
  new_values.append((calendar.month_name[int(value[1])] + value[0], value[2]))

values = new_values
```

Figure 4.13.: Dialog for creating custom tasks.


Info

Analysis failed.

OperationalError: no such column: a.i (Task ID: 115257198)

Figure 4.14.: Error message example.

4.7.3. Execution

The execution works similar to the one of the general analysis. A queue of jobs is created, which contains the chart models of the activated tasks. This queue is processed by a set of threads. The difference to the general analysis is that there are no predefined SQL Select statements. Instead, the custom SQL Select statement, the user configured via the dialog, is used.

4.8. Error Handling

Recoverability is an essential non-functional requirement of our tool, meaning that in the case of an error, the application should not crash. Instead, the user should be informed about the error and if possible how to solve it. Meaningful error indications are especially important when executing custom tasks as a SQL Select statement or user-written Python code may contain errors. Moreover, the response from a SQL Select statement may not have enough columns to fit the data requirements of the selected chart type. However, also collecting data may fail, for example, when the internet connection is lost during the collection process. Errors are visualized via custom message dialog windows, where an example can be seen in Figure 4.14. The message corresponds to an error, which occurred when a custom task with an invalid SQL Select statement has been executed. It shows a short description of the error and the ID of the failed task. A lot of tasks are handled by worker threads. However, only the main thread (user interface thread) can issue messages dialog windows. As a result, errors are passed from the worker threads to the main thread. This is done using custom signals. In the case of an error, a signal is emitted containing a description of the error and further information. A model that created a worker thread always listens to their signals and propagates the received messages back to its corresponding controller using custom signals. Finally, the controller knows how to resolve the error and creates a message dialog window to inform the user about the error.

4.9. Distribution

To distribute the application to the end users, the source code has to be built and packaged together with the required resource files. To make the packaging process easier, the Qt resource system[17] is used, which allows bundling resources like images or style sheets in Python files. The advantage is that the resources are made independent of the different paths on different platforms. It works by creating a .qrc file and defining which resources should be loaded and bundled into a python file (Fitzpatrick, 2020). An excerpt from our resource file can be seen in Listing 4.7. Each resource has a prefix and is defined via an alias name. In the case of the saving icon with the path *../Images/Icons/save.png* is given the alias name *save.png* and has the prefix *icons*. The purpose of the prefix is to group resources. To create a Python file containing the data from the resources defined in the .qrc file, the command *pyrcc5 my_resources.qrc -o my_resources.py* is executed. The created Python file with the name *my_resources.py* can be important like any other Python file. To access a resource, the path to a resource is not required any longer. Instead, it can be replaced with *:/prefix/alias*. Concerning the saving icon, the resource path is *:/icons/save.png*.

```
<!DOCTYPE RCC>
<RCC version="1.0">
  <qresource prefix="icons">
    <file alias="save.png">../Images/Icons/save.png</file>
    <file alias="open.png">../Images/Icons/open.png</file>
```

[17]https://doc.qt.io/qt-5/resources.html

```
  </qresource>
  <qresource prefix="qss">
    <file alias="menu_button.qss">../StyleSheets/menu_button.
   css</file>
  </qresource>
</RCC>
```

Listing 4.7: QRC file example.

To finally create a stand-alone executable, we use the package PyInstaller[18] that supports PyQt5 applications. PyInstaller builds and bundles the source code and resources into a single package, which means that the user does not need to install a Python interpreter on the machine. Instead, a Python interpreter and the required modules are part of the created package. For correctly building and packaging our application, a .spec file is created, which can be passed to PyInstaller as a command-line argument. The file contains instructions on how the applications should be built. Moreover, it allows to define a desktop icon, exclude libraries to reduce the package size, include additional resources which are not part of the resource system, and give the application a name. When the .spec file has been executed with PyInstaller via the command *pyinstaller my_spec.spec*, a distribution folder containing the application executable is created automatically (Fitzpatrick, 2020). Finally, the whole distribution folder can be zipped and distributed to the end-user. After the zipped file is unpacked, the application can be started and should work out of the box.

4.10. Summary

This chapter provided deep insights into the implementation of the tool for collecting and analyzing game data and reviews from the platform Steam. It described the layout and style of the user interface and lists a set of custom widgets. Concerning the layout, each main aspect of the application was given a separate view, which can be switched via menu buttons. To give the tool a modern look, a dark style from a third-party library was

[18]https://www.pyinstaller.org/

chosen, and extended by additional elements. Moreover, custom widgets were implemented for reusability and to fasten the development process.

For storing and associating the collected data, an SQLite database was set up via an SQL schema. The database was built to support built-in mathematical functions and have text-based search capabilities.

The next part described the architecture for collecting and analyzing data. Simplified, it is a combination of the worker thread pattern and the HMVC pattern. The implemented architecture allowed us to completely separate the data models from their presentations. Furthermore, outsourcing specific tasks to worker threads avoids that the user interface blocks while long-running tasks are executed. Although the chosen architecture lead to a heavy increase of the development time, it paid off, as it provided us with a well to maintain code and a non-blocking user interface.

Another part of the chapter focused on the implementation for collecting general game data and reviews from Steam. An in-tool configurator was integrated that allows narrowing the data collection to a set of games fulfilling specific characteristics. The procedure for collecting game IDs and general game data is handled by a separate thread, where data is collected by crawling the Steam search page and the individual game pages. To collect reviews, the Steam Web API is utilized, which allows requesting reviews for a specific game. To visualize the collected data, different chart model and view classes as well as a table model and view class were created. The model classes contain the business logic to request and prepare data for the visualization via Matplotlib charts or PyQt5 tables. The view classes have the actual implementation for presenting the data of the chart models and are responsible for creating the user interface elements, i.e., the presentation of the models.

The chart and table classes are used as part of the implementation for performing general and custom analysis tasks. In the case of the general analysis, predefined charts are created and updated on demand. In the case of the custom analysis, a dialog was implemented that allows users to define the properties of a chart or table and what data it should visualize.

As recoverability is an essential requirement of our application, one part of the chapter had a look at error handling. The errors are visualized via message dialog windows that contain a meaningful description of the error.

Finally, the last part provided a possibility to build and distribute the tool to the user using the Python package PyInstaller. Furthermore, it described how resources like images or text files can be bundled to make resources independent of the file location.

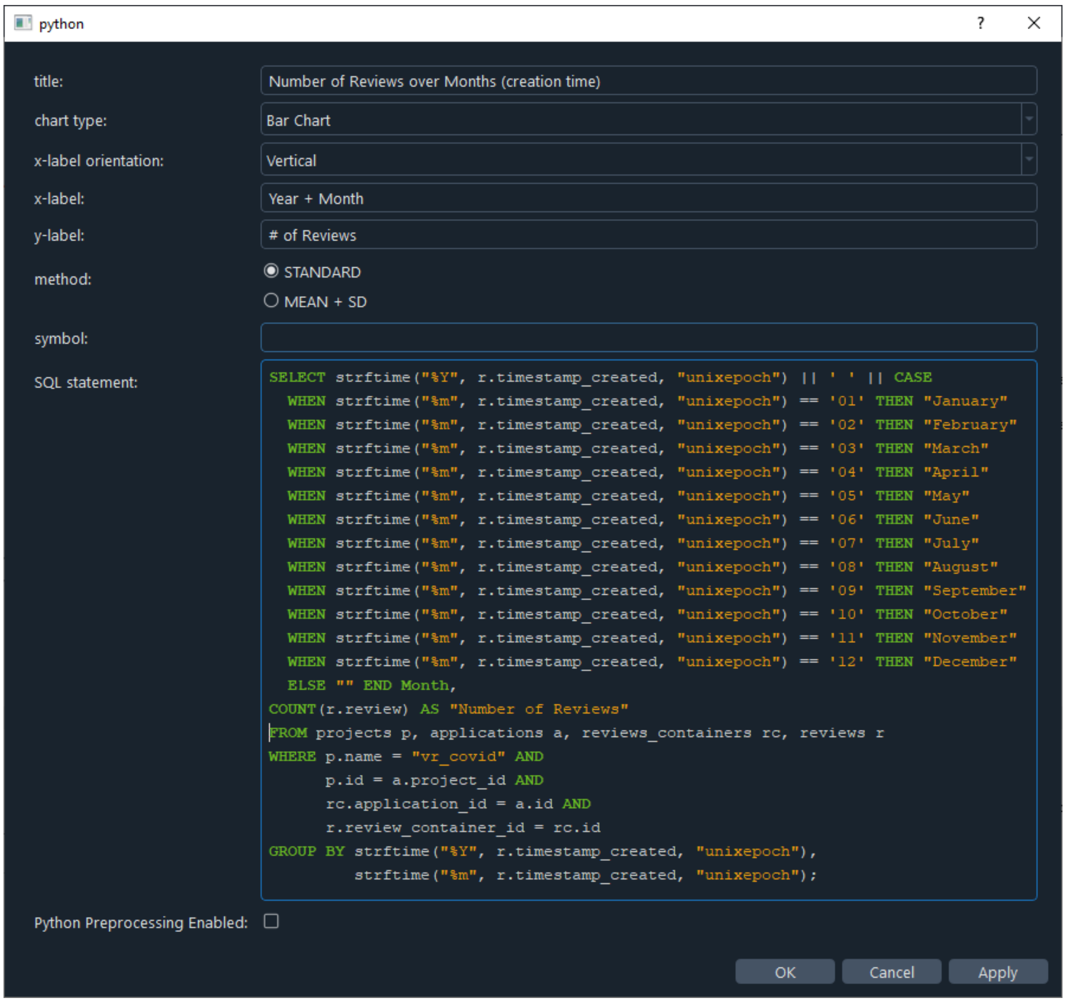
5. Example Use Case

In a currently ongoing study, we utilize the tool to investigate how the Covid-19 pandemic influences the user's decision to play Virtual Reality games published on Steam. Furthermore, we analyze the impact of the coronavirus on the written reviews and try to find connections to the virus. To demonstrate the capabilities of our tool, we discuss one of the study's research questions. In particular, we discuss the research question "Do the users write more VR game reviews during the pandemic?". To answer the question, we collected game data and reviews of Virtual Reality games available on Steam. A summary of the dataset can be seen in Table 5.1.

# of games	4241
# of reviews	386,705
review language(s)	english
review type	all (positive and negative)
purchase type	all (free and non-free games)
retrieval date	1st June 2021
Steam search URL	https://store.steampowered.com/search/?sort_by=Name_ASC&category1=998%2C994&vrsupport=401

Table 5.1.: Dataset description.

To investigate the number of reviews written over time, two custom analysis tasks were created. The first one visualizes the number of reviews over years, and the second one the reviews over months. For both, a bar chart was chosen as the chart type. For the monthly-based task, the corresponding custom analysis dialog with the SQL Select statement is shown in Figure 5.1. As can be seen, the conversion from the month value to the month name is done in the SQL Select statement. Optionally, the same can be achieved by manipulating the SQL result via Python.



title: Number of Reviews over Months (creation time)

chart type: Bar Chart

x-label orientation: Vertical

x-label: Year + Month

y-label: # of Reviews

method: [✓ STANDARD] MEAN + SD

symbol:

SQL statement:

```
SELECT strftime("%Y", r.timestamp_created, "unixepoch") || ' ' || CASE
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '01' THEN "January"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '02' THEN "February"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '03' THEN "March"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '04' THEN "April"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '05' THEN "May"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '06' THEN "June"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '07' THEN "July"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '08' THEN "August"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '09' THEN "September"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '10' THEN "October"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '11' THEN "November"
  WHEN strftime("%m", r.timestamp_created, "unixepoch") == '12' THEN "December"
  ELSE "" END Month,
COUNT(r.review) AS "Number of Reviews"
FROM projects p, applications a, reviews_containers rc, reviews r
WHERE p.name = "vr_covid" AND
      p.id = a.project_id AND
      rc.application_id = a.id AND
      r.review_container_id = rc.id
GROUP BY strftime("%Y", r.timestamp_created, "unixepoch"),
         strftime("%m", r.timestamp_created, "unixepoch");
```

Python Preprocessing Enabled: ☐

Figure 5.1.: Custom analysis dialog for visualizing the number of written reviews over months.

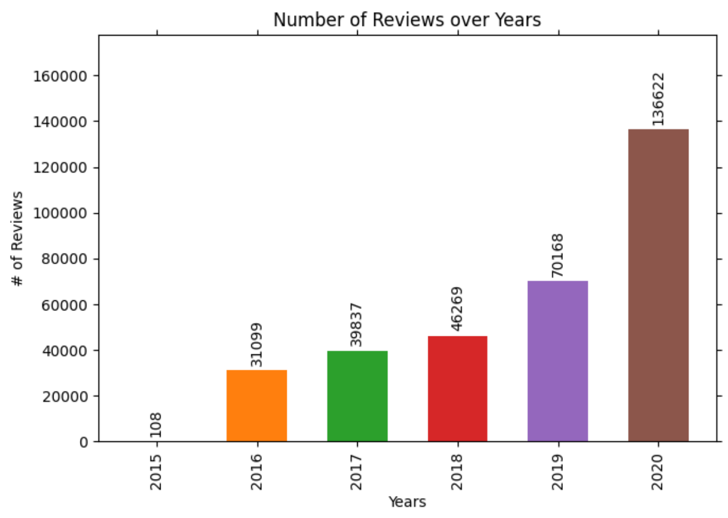


Figure 5.2.: Number of written reviews over years.

Consequently, we present the current results of the research question with visualizations that have been created with our implemented tool.

RQ1: Do the users write more reviews during the pandemic?
As can be seen in Figure 5.2 the number of reviews written in the year 2020 increased by approximately 94% compared to the year 2019. The year 2020 corresponds to the year in which the coronavirus mainly occurred. Although a rise can also be detected in the previous years (∼28%(2016-2017), ∼16%(2017-2018), ∼51%(2018-2019)), the change from 2019 to 2020 is much higher. By looking at the figure, March 2020 stands out. By investigating what is special about this month, we found out that a large number of worldwide countries went into a national or localized lockdown starting within March 2020 (BBC, 2020).

Although the study is still in an early stage, the tool has already proved useful to collect and analyze data from Steam. So far, the tool has provided us with a way to get a quick overview of the collected dataset. Furthermore, the definition of a custom analysis inside the tool allowed us to answer the discussed research question.

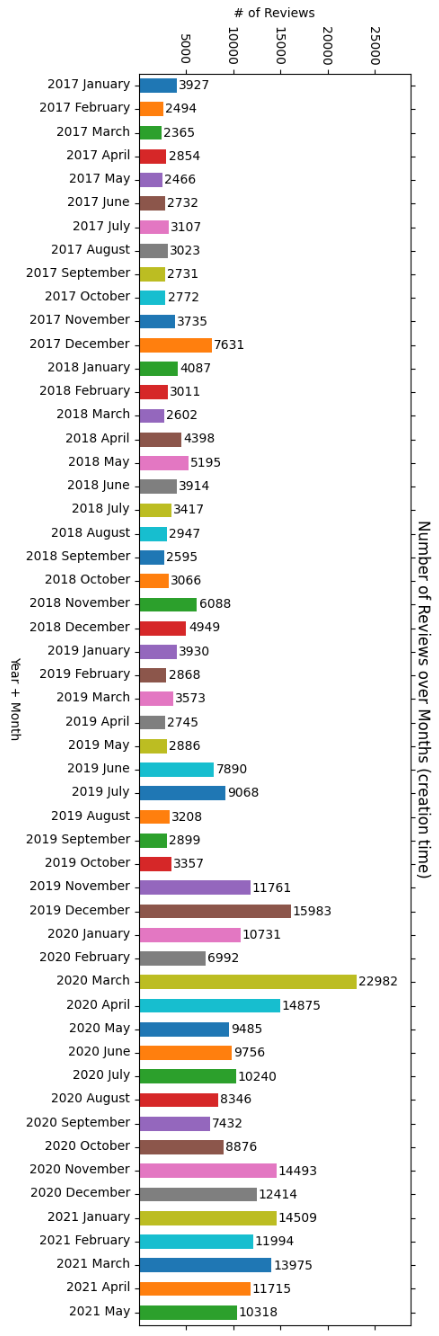


Figure 5.3.: Number of written reviews over months.

6. Evaluation

To obtain feedback on our implemented tool, a first study with six domain experts was conducted. The aim was to gather valuable information from the criticisms and reactions of the participants and use the gained insights to improve the tool. For the reason that the tool is supposed to be used by game researchers with prior Computer Science knowledge, we decided to restrict the study to participants that have Computer Science knowledge as well. Moreover, we argue this decision by the fact that people, who do not have any experience with a programming language or SQL, are not able to exhaust the full potential of our tool without doing a lot of prior research.

Via the results of the study, we wanted to achieve the following points:

- Assess the usability of the tool.
- Assess the emotions participants have during the usage of the tool.
- Detect and rate the importance of missing tool features.
- Detect errors and bugs.

6.1. Procedure & Material

The study was conducted online. Beforehand the participants received a unique identification number and a PDF file containing the instructions. We allowed the participants to take the study at any time within a one-week time range. The only limitation was that they had to announce their choice beforehand, enabling us to be available for questions in the case of problems. In the study, the participants had to perform several activities, starting with downloading a zip file containing the tool and additional required files from Google drive. In the course of the study, the participants had to fill out three different questionnaires, namely the (1) pre-questionnaire, (2) tasks-questionnaire, and the (3) post-questionnaire, where the questions can be found in the Appendix A. During the download, the participants could already start filling out the pre-questionnaire.

6.1.1. Pre-Questionnaire

The purpose of the pre-questionnaire was to obtain general information about the participants and their experience concerning computers, video games, game distribution platforms, game analysis, and data analysis. Concerning the knowledge in the usage of computer games and video games, the participants had to rate their knowledge on a 5-point Likert scale between (1) *strongly agree* and (5) *strongly disagree*. For other experience-related questions, yes-no questions were formulated. If the yes-option was selected, the participants had to describe their experience in the form of an open text.

6.1.2. Tasks-Questionnaire

Having the pre-questionnaire finished, the participants were instructed to run the tool and familiarize themselves with it for about 5 minutes. Next, they had to answer a set of questions in the tasks-questionnaire with the help of the tool. The questionnaire was structured into three question groups having the names *Analysis*, *Custom Analysis*, and *Data Collection*, each focusing on another main functionality of the tool. To allow performing a general or custom analysis within the tool, a dataset was collected beforehand. The project associated with the dataset was given the name *example* and contained game and review data of Virtual Reality games available on Steam. In addition, at the beginning of each question group, a short textual description of the functionality and further instructions were provided.

6.1.3. Post-Questionnaire

The last activity was to fill out the post-questionnaire, where the users had to give feedback on the tool based on their perception while performing the tasks of the tasks-questionnaire.

Overall Impression

First, the participants had to rate the tool on a 5-point Likert scale between (1) *not at all* and (5) *very much*. In addition, they had to describe what they liked about the tool in the form of an open text. Similarly, they had to describe what they did not like. Furthermore, they were asked if they observed any bugs in the system. If this was the case, they had to detail their observation. Moreover, the participants were asked if they had any problems or difficulties while solving the tasks of the tasks-questionnaire.

Features & Improvements

Another part of the questionnaire focused on missing features and potential improvements. For a set of features, the participants had to rate if they wanted to see them in a later release. For the rating, a 5-point Likert Scale between (1) *not at all* and (5) *very much*, was used. Furthermore, the participants could suggest features on their own via a textual description. Similarly, the participants were encouraged to provide ideas for further improvements.

Computer Emotion Scale (CES)

We measured the emotions of the participants while performing the tasks of tasks-questionnaire using the computer emotion scale (CES) introduced by Kay and Loverock (2008). Initially, it was created to assess users' emotions while learning new software. It consists of twelve feelings, which the users have to rate depending on how long they felt the respective emotion. The rating is a 4-point Likert scale with the values (0) *None of the time*, (1) *Some* *of the time*, (2) *Most of the time*, and (3) *All of the time*. Each of the feelings belongs to one of the four groups, happiness, sadness, anxiety, or anger, that are evaluated.

System Usability Scale (SUS)

For assessing the usability of the tool, the system usability scale (SUS) from Brooke (2020) was used. The SUS is known to be a quick and cheap solution for the usability assessment of a system. It consists of ten items a user has to rate on a 5-point Likert scale between (1) strongly disagree and (5) strongly agree. For calculating the SUS score, the rating of each even item is subtracted from the value 5. In contrast, the ratings of each odd item is decreased by 1. The sum of the modified item ratings multiplied by 2.5 corresponds to the final SUS score. To evaluate the rating, we used the interpretation of Bangor et al. (2009), who introduced an adjective scale, which correlates well with the score of the SUS.

6.2. Participants

As previously mentioned, the study focused on participants with prior Computer Science knowledge. As a result, all the participants rated their computer knowledge very high. Each participant has a bachelor's or master's degree in Computer Science. In total, six people (1 female, 5 male) participated in the study. All have already heard of the game distribution platform Steam and have already bought a game on it. In addition, half of them have already written a review on Steam. Furthermore, half of the participants have experience with collecting data from Steam, where one noted that the Steam Web API was utilized for this purpose. However, only two of the three participants have experience with analyzing the collected data. Concerning data analysis not restricted to Steam, four participants have experience, where all of them mentioned Python and its libraries as the used technology.

6.3. Results

The results of the questionnaires were evaluated in different aspects, starting with the overall impression.

6.3.1. Overall Impression

The feedback on the tool was quite positive. The worst rating of the 5-point Likert scale is 3, and the best 5 (M = 4.00, SD = 0.63). One positive feedback statement that stands out is the appealing design of the graphical user interface, which was explicitly mentioned by three of the six participants. In addition, two participants value the configurability of the tool. Other positive feedback concerns the tool's capability to run tasks in the background and the uncomplicated usage. However, one negative feedback was that it takes some time to get used to the tool. Other negative feedback refers to suggested improvements, missing features, or inconsistencies, discussed in the next part.

6.3.2. Features & Improvements

As previously mentioned, the participants were asked to rate the importance of features that are not implemented so far based on the question, if they wanted to see the feature in the next release of the tool. In total, the participants had to rate seven features on a 5-point Likert scale between (1) *not at all* and (5) *very much*. Table 6.1 shows the mean score and the standard deviation for each feature. As can be seen, the in-tool documentation is the most wanted feature, followed by a possibility to save all charts at once. The feature that takes last place is an option to provide results in a text field in which output, resulting from a custom analysis task, should be shown in the form of a string.

In addition, the participants were encouraged to provide their own feature suggestions and tool improvements. The following list provides a summary of their suggestions:

Feature Description	**Mean**	**SD**
In-tool documentation	5.00	0.00
Possibility to save all charts and tables at once	4.50	0.55
Possibility to create custom chart designs inside the tool (e.g., contour chart)	3.83	1.47
Option to select which data should be collected from Steam (to save memory)	3.67	1.37
Possibility to change the visibility order of charts/tasks	3.33	1.03
Tabs to structure custom analysis tasks	3.17	1.17
Text-only output (result is converted to a string and visualized in a text field)	2.33	1.21

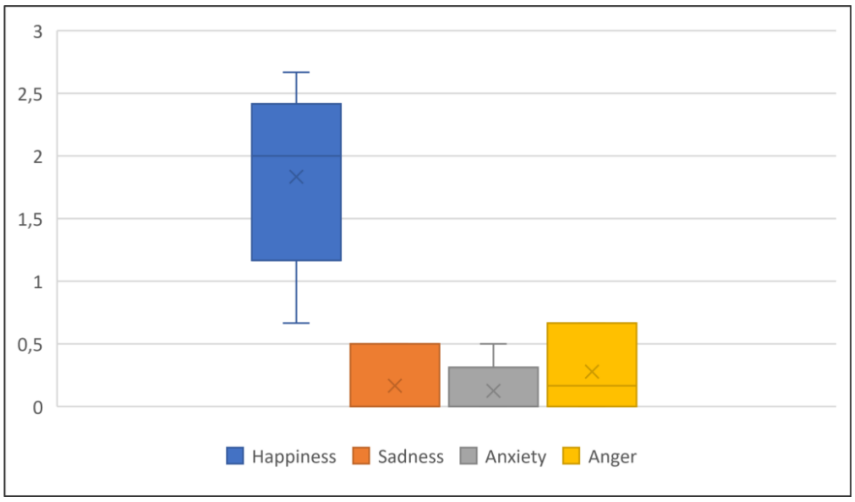
Table 6.1.: Rating results based on the users' opinions on the usefulness of non-implemented features. The corresponding question asked the users if they wanted the see the feature in a later release of the tool. For the rating, a 5-point Likert scale between (1) *not at all* and (5) *very much* was used.

.

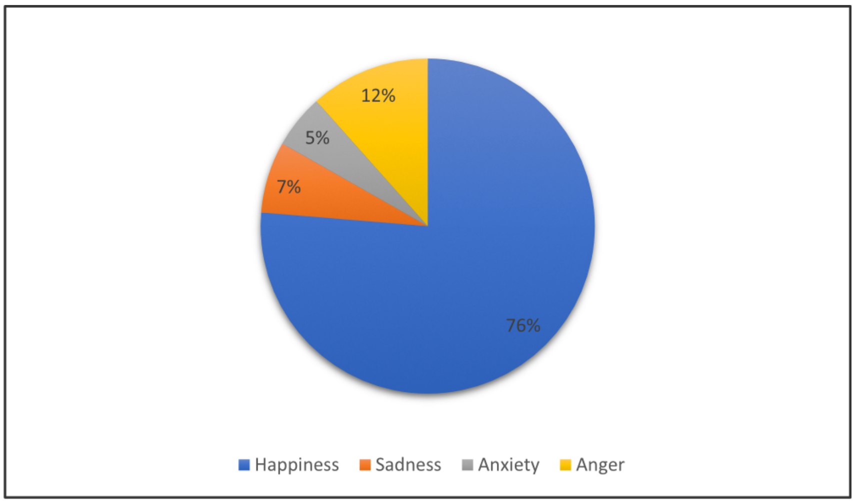
- Additional checkboxes for selecting all genres, tags, or game area details, at once.
- Option to switch the dark theme style of the tool to a light theme style.
- Possibility to debug the Python code entered in custom analysis dialogs.
- Visual feedback when a collection process finished.

Furthermore, the participants had to describe problems, difficulties, bugs, and things they did not like about the tool in separate questions. Consequently, we aggregated the information of the individual questions and created the following list:

- The *Ok* button of the custom analysis dialog does not apply changes like the *Apply* button. Instead, it behaves like the *Cancel* button.
- The initial size of the custom analysis dialog is too small.
- On 4k monitors, the right sidebar is too small, and the window cannot be fully expanded.



(a) Visualization of the CES results.



(b) Composition of feelings.

Figure 6.1.: CES results.

6.3.3. Emotions

The emotion the participants claimed to feel the longest, is happiness (M = 1.88, SD = 0.72), followed by anger (M = 0.28, SD = 0.33), sadness (M = 0.17, SD = 0.26), and anxiety (M = 0.125, SD = 0.21). The results are shown in the form of a boxplot in Figure 6.1a. By converting the overall feelings to the composition visualized in Figure 6.1b, with the assumption that a person can only feel one feeling at a time, shows that the participants felt happy most of the time. Investigating the three items that build the happiness group (satisfaction, excitement, and curiosity), satisfaction (M = 2.17, SD = 0.75) was felt the longest, followed by curiosity (M = 1.83, SD = 1.17) and excitement (M = 1.5, SD = 0.83). Concerning the individual emotions of the angry group, none of the participants felt angry (M = 0, SD = 0), but sometimes irritable (M = 0.5, SD = 0.55) or frustrated (M = 0.33, SD = 0.52). Throughout the whole study, none of the participants felt angry, nervous, anxious, or helpless at all.

6.3.4. Usability

The achieved mean score of the system usability scale was 77.08 (SD = 12.98). Figure 6.2 shows a visualization of the mean result and the standard deviation and contains the adjective rating scales of Bangor et al. (2009). According to the adjective scale, the tool has a good to excellent usability. The worst individual SUS score is 65, which indicates an OK to good usability. In contrast, the best individual SUS score is 100, which equals the best imaginable usability with respect to the adjective scale.

6.3.5. Tasks Success Rate

As previously mentioned, the tasks-questionnaire was structured into three question groups, namely *Analysis*, *Custom Analysis*, and *Data Collection*, each focusing on another main functionality of the tool. The activities the participants had to perform in the tool to be able to answer the questions seemed to be of the appropriate difficulty. All the participants gave the

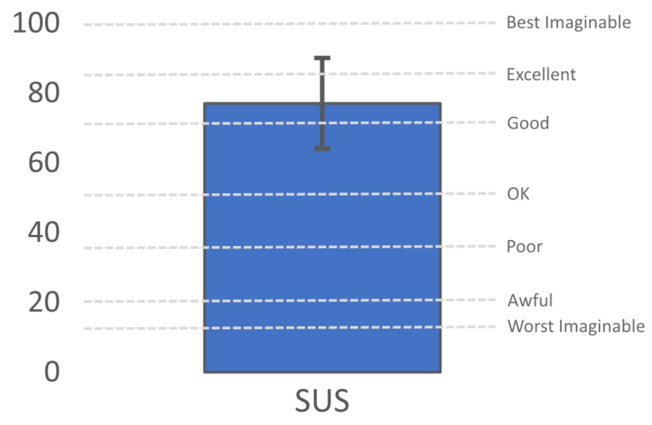


Figure 6.2.: Visualization of the SUS with adjective rating lines of Bangor et al. (2009).

correct answer to all questions of the question groups *Analysis*, and *Data Collection*. Concerning the *Custom Analysis* group, only one person answered two questions incorrectly.

6.4. Discussion

The feedback of the participants was quite positive. The results of the computer emotion scale show that the users felt happy most of the time. Concerning the SUS, the mean score (M = 77.08, SD = 12.98) indicates good to excellent usability. Furthermore, the tool was valued for its appealing design and ability to execute tasks in the background. Moreover, they were impressed by the configuration capabilities of the tool. Still, applying the suggested improvements of the participants might further improve the usability. Especially, adding checkboxes for selecting all genres, tags, or game area details at once might have a big impact on the usability as the majority of participants missed the feature in the tool. One bug or inconsistency that was detected by two participants was that the *Ok* button of the custom analysis dialog does not apply changes like the *Apply* button. Instead, the button behaves like the *Cancel* button. The bug might have a negative impact on the users' emotions during using the tool as they might have to perform the same actions several times before realizing the bug.

Three participants explicitly mentioned that the tool is easy to use. However, all participants want to see an in-tool documentation in the next version of the tool. The reason for this might be the high number of available configuration options. Although the tool provides additional information using tooltips, understanding all options without detailed documentation or provided examples might not be feasible.

The study gave us a good opportunity to obtain feedback on the tool's main functionalities. We could derive a set of improvements and features that might enhance the tool. Furthermore, we could determine bugs and inconsistencies within the system.

7. Lessons Learned

This chapter discusses the experience gathered during the creation of this work. It is divided into three parts. The first focuses on the background research, the second on the development, and the last one on the conducted study.

7.1. Theory

The background research shows that data from Metacritic or Steam has been exploited in different ways and to achieve various goals. The main topics concern the helpfulness of reviews, review sentiment, the correlation between different rating types, the influence of reviews on video game success, the playability of video games, and recommendation systems. An investigation of the studies revealed that the methodologies defined by the authors are similar concerning the Steam data sources and the way how the data is stored. Consequently, different approaches for collecting and storing data from Steam were identified. Based on the research, we chose the approach that was best suited for our tool. Furthermore, looking at different analytic tools supplied us with ideas for designing our tool's analysis functionalities.

7.2. Development

For the development of our tool, we used the programming language Python. The choice turned out to be the right one. Collecting game data with the Scrapy library and the reviews with request library, a standard

Python library, were easy to achieve. Moreover, the language proved to be well suited for creating charts. Python is further easy to learn compared to other languages like C++. As a result, the language was also a good choice for the custom analysis functionality of the tool, where users can manipulate and analyze data via Python.

The architecture helped us to make the tool efficient and its user interface non-blocking when long-running tasks are executed. Furthermore, the Hierarchical-Model-View-Controller (HMVC) pattern provided us with a well maintainable project structure. The main reason is that we have a complete separation between the business logic and the presentation of the data. Thus, applying changes to the view can be done without affecting the business logic. With the implementation of the HMVC pattern, we faced the problem of an increased development time. However, it paid off as the resulting codebase is well structured and well to maintain. In addition, the worker thread pattern allows running analysis tasks or collection tasks completely in the background. Consequently, users can perform several activities in the tool at the same time. For example, collecting and analyzing data can be done simultaneously. The biggest challenge was to implement the message exchange of the worker threads with the user interface thread. Especially errors needed to be handled correctly to avoid faulty application states.

7.3. Evaluation

From the evaluation, we obtained valuable feedback on the current state of our tool. The usability of the tool is deemed to be good to excellent. Furthermore, the users felt happy most of the time while using the tool. Still, the users provided us with ideas about additional or missing features and improvements that might further enhance the tool. These features and improvements are discussed in the following chapter.

8. Future Work

Although the results of the conducted study indicate good usability, further work is necessary to get the best out of the tool. The evaluation revealed improvements and missing features that can enhance the usability of the tool or can have a positive impact on the feelings of the users. Thus, this chapter gives an outline of the work done in the upcoming iteration of the development cycle.

8.1. Improvements

The participants only suggested a minor number of improvements to enhance the tool. Within the tool, only a small number of bugs or misconceptions were detected and described by the users.

8.1.1. Bug Fixes

One problem that occurred is that the *OK* button of the custom analysis dialog does not work as expected. It should apply changes made in the dialog like the *Apply* button and, in addition, close the dialog. However, the *OK* button behaves like the *Cancel* button. The misconception can be fixed by calling the same method that is called when the *Apply* button is clicked. Still, fixing the bug might have a big impact on the tool's usability, and might lead to a decrease of negative feelings while the software is used.

Another problem that occurred is that some user interface elements are too small on 4k monitors. Thus, more testing on monitors with different resolutions is necessary. The goal is to make the user interface completely independent or responsive to a monitor's resolution.

A further technical improvement that was not mentioned by any participant is to make the tool more efficient via multiprocessing.

8.1.2. Multiprocessing

As mentioned in the design chapter, Python has the problem of the global interpreter lock (GIL). This lock has the effect that only one instruction within the same process is executed at a time. The GIL is released in specific cases like when loading a web resource, reading a file, or accessing a database. However, using multiple threads in the case of CPU-heavy tasks might be less performant compared to a single-threaded environment. To overcome the problem, multiprocessing is one possible solution. A separate process is already utilized for executing user-written Python code. Nevertheless, outsourcing more work to separate processes might further improve the performance of the tool. However, the additional overhead of creating processes might be worse than just running the work in the current thread. Thus, it is necessary to compare the performance of the different approaches.

8.2. Tool Features

Aside from suggesting improvements, participants also had to rate the importance of features that are not implemented so far, based on their opinion. Furthermore, they were encouraged to suggest further features via free text. Consequently, we discuss a subset of features that should be part of the next version of the tool.

8.2.1. In-Tool Documentation

The highest rated feature was the in-tool documentation. One possibility to add this feature is to create a static documentation written in HTML. The used GUI framework of our tool (PyQt5) allows to display HTML content via the *QWebEngineView*[1]. One idea is to place an additional button in tool, which opens a separate dialog that contains the HTML documentation. One requirement is that the button should be easy to find.

8.2.2. Custom Chart Types

Another feature that achieved a high rating is a possibility to define custom chart design inside the tool. One possible approach is to add an additional menu point inside the tool. Inside the new view associated with the new menu point, users should be able to create Matplotlib charts via Python by following specific rules. After a new chart type has been created, it should be possible to use it in the custom analysis dialog.

8.2.3. Data Selection

The feature that takes the fourth place in the ranking is an option to select which data fields should be collected from Steam. The idea behinds this is to avoid the storing of data that is not required by the user. One approach is to implement a custom SQL table creator, where tables are created for each individual project taking the selected data fields into account. For example, if the user only wants to save the title of a game, the *applications* table should only contain a field to store the game title.

8.2.4. Python Debugging

One feature that was suggested by a participant, is a possibility to debug the Python code written in a custom analysis dialog. Our first step concerning Python debugging will be to add a convenient way to access the output written to the *stdout*, *stderr* descriptors. This can be achieved by adding an additional text area per task, where Python code is used. A possible position

[1]https://doc.qt.io/qt-5/qwebengineview.html

of the text area would be directly below the visualization of the individual charts or tables in the custom analysis view.

8.2.5. All Checkboxes

One feature that most of the participants missed within the tool is the option to select all genres, tags, or game area details checkboxes at once in the view associated with the general analysis. They mentioned that the feature would save them a lot of time as they would not have to click each checkbox individually. As a result, this feature will be part of the next version of the tool.

8.3. User Study

At the end of the upcoming review cycle, we plan to conduct a further user study with more participants (approx. 30). In the first iteration, the small number of six participants was appropriate as the main aim was to detect the main issues in the system. In the case of our tool, the majority of the participants wrote about the missing *all* checkboxes for selecting all genres, tags, or game area details. Consequently, the participants might not have mentioned other problems they observed while using the system. However, having the main problems fixed, it makes sense to increase the number of participants to assess the tool's usability with higher accuracy. A higher number of participants might also help us detect further improvements or other problems within the software.

9. Conclusion

A rapid growth of the video game industry over the last few years can be observed. Game distribution platforms like Steam allow buying and downloading games online. Furthermore, Steam and other platforms like Metacritic enable people to rate games with an optional textual description. As a result, rising interest among game researchers and developers in video game distribution and review platforms can be detected. The data of Steam has been exploited using different approaches to achieve various goals. However, an overlap of the researchers' methodologies concerning the collection and the analysis of data exists.

This thesis introduced a tool that should ease the process for collecting and analyzing data from Steam. It presented a design and concept defining the objective, user target group, requirements, and logical architecture. Moreover, a detailed implementation was provided that gives deep insights into the different functionalities added to the tool. Support for collecting general game data and reviews was implemented, where the data can be additionally narrowed based on specific attributes. For analyzing the collected data, analytic capabilities were added enabling users to obtain a general overview of a dataset or to create custom analysis via SQL Select queries or Python code on their own.

An example use case in the form of an ongoing study was described, where the tool was utilized for collecting and analyzing data. The tool proved useful and satisfied all our needs. Furthermore, a small user study was conducted to get feedback on the tool and detect problems or inconsistencies within the system. In addition, the tool's usability and users' emotions while using the tool were assessed as part of the study. The results indicate good to excellent usability and show that the participants felt happy most of the time. Concerning features and improvements, a lot of derivations could be made from the results. Overall the participants liked the tool and would use it in the case they want to collect and analyze data from the game distribution platform Steam.

To summarize, it is crucial for game developers and game researchers to understand the dynamics and settings of the video game industry. Thus, we believe that our tool is one step in the right direction as it allows obtaining deep insights into games and player behavior.

Appendix

Appendix A. Questionnaires

A.1. Pre-Questionnaire

Personal Information

#	Question	Answer Type
1	Please provide your identification number.	Short free text
2	Gender	Female, Male, Other
3	Profession	Student, Employed, Unemployed or Other
4	Field of Study	Short free text
5	Job title	Short free text
6	Highest level of education	Short free text

General Questions

#	Question	Answer Type
1	I have heard of the game distribution platform Steam.	Yes/No
2	I am an expert in usage of computers.	1 not at all - 5 fully agree
3	I am an expert in usage of video games.	1 not at all - 5 fully agree
4	I am familiar with the game distribution platform Steam.	1 not at all - 5 fully agree
5	I have already bought a game on Steam.	Yes/No
6	I have already written a review on a game distribution or review platform.	Yes/No
7	I have already written a review on Steam.	Yes/No
8	I have experience on collecting game or review data from Steam.	Yes/No
9	Concerning the previous question, describe your experience.	Long free text
10	I already have experience on analyzing game or review data collected from Steam.	Yes/No
11	Concerning the previous question, describe your experience.	Long free text
12	Do you have any other experience with data analysis?	Yes/No
13	Concerning the previous question, describe your experience.	Long free text
14	Concerning the previous question, which tools did you use?	Long free text

A.2. Tasks-Questionnaire

A.2.1. Analysis

#	Question	Answer Type
1	How many games have the Steam rating "Positive"?	Number-only text
2	How many games were released in the year 2019?	Number-only text
3	Which year (between 2017 and 2020) has the highest average character count of all (positive + negative) reviews?	Number-only text
4	How many games have the genre "Adventure" assigned?	Number-only text
5	Which genre is most often assigned to games?	Short free text
6	What is the name of the Head-Mounted Display (HMD) that supports the most games?	Short free text
7	How many games with the genre "Indie" are supported by "Oculus Rift"?	Number-only text

A.2.2. Custom Analysis

#	Question	Answer Type
1	How many reviews of the project with the name "example" were written in the year 2020?	Number-only text
2	How many reviews of the project with the name "example" were written in the month April 2020?	Number-only text
3	How many games of the project with the name "example" have $>$ 10 reviews?	Number-only text
4	How many games does the project "example" have?	Number-only text
5	The chart with the ID: 256264100 has two errors and is currently deactivated. Try to fix it and report the number of reviews for the genre "Indie".	Number-only text
6	The task with the ID: 6653165 has the same SQL Select statement as the first one (ID: 115257198). However, each value of the "Number of Reviews" value should be multiplied by 9 via Python. If you are not familiar with Python you can skip the question. In this case, write -1 as the answer. If you are familiar with Python but cannot solve the task, write -2 as the answer. If you could solve it, write the new "Number of Reviews" value for the year 2017 as the answer. Give yourself a time limit of 10 minutes.	Number-only text

A.2.3. Data Collection

#	Question	Answer Type
1	What is the Steam link of your created project?	Short free text
2	What is the review filter type of your project?	Short free text
3	What is the review language of your project?	Short free text

A.3. Post-Questionnaire

A.3.1. Overall Impression

#	Question	Answer Type
1	Please provide your identification number.	Short free text
2	How did you like the tool?	1 not at all - 5 very much
3	What did you like?	Long free text
4	What did you not like?	Long free text
5	Did you observe any bugs?	Yes/No
6	Concerning the question above, provide a description of your observed bugs.	Long free text
7	Did you have any problems while solving the tasks?	Yes/No
8	Concerning the question above, describe your problems.	Long free text
9	Did you have any problems or difficulties while solving the tasks?	Yes/No
10	Concerning the question above, describe your problems and difficulties.	Long free text
11	Do you want to see the following features in a later release of the tool?	Sub-question Array
11.1	Tabs to structure custom analysis tasks	1 not at all - 5 very much
11.2	Possibility to create custom chart designs inside the tool (e.g., contour chart)	1 not at all - 5 very much
11.3	Option to select which data should be collected from Steam (to save memory)	1 not at all - 5 very much
11.4	Possibility to save all charts and tables at once	1 not at all - 5 very much
11.5	Possibility to change the visibility order of charts/tasks	1 not at all - 5 very much
11.6	In-tool documentation	1 not at all - 5 very much
11.7	Text-only output (SQL result or Python result is converted to a string and visualized in a text-field)	1 not at all - 5 very much
12	Do you have any other suggestions of features that you want to see in a later release of the tool?	Yes/No
13	Concerning the question above, describe your feature suggestions.	Long free text
14	Would you use the tool to study Steam data and reviews?	Yes/No
15	Provide suggestions to improve the tool (e.g., user interface, design, etc.).	Long free text
16	Describe potential use cases the tool might prove useful.	Long free text

A.3.2. System Usability Scale (Brooke et al., 1996) (1 strongly disagree - 5 strongly agree)

#	Item
1	I think that I would like to use this system frequently
2	I found the system unnecessarily complex
3	I thought the system was easy to use
4	I think that I would need the support of a technical person to be able to use this system
5	I found the various functions in the system were well integrated
6	I thought there was too much inconsistency in the system
7	I would imagine that most people would learn to use this system very quickly
8	I found the system very cumbersome to use
9	I felt very confident using the system
10	I needed to learn a lot of things before I could get going with this system

A.3.3. Computer Emotion Scale (Kay & Loverock, 2008) (0 none of the time- 3 all of the time)

#	Item	Group
1	Satisfied	Happiness
2	Disheartened	Sadness
3	Anxious	Anxiety
4	Irritable	Anger
5	Excited	Happiness
6	Dispirited	Sadness
7	Insecure	Anxiety
8	Frustrated	Anger
9	Curious	Happiness
10	Helpless	Anxiety
11	Nervous	Anxiety
12	Angry	Anger

Bibliography

Ajitsaria, A. (2017). *What is the python global interpreter lock (gil)?* Retrieved November 16, 2021, from https://realpython.com/python-gil/. (Cit. on p. 52)

AskPython. (2021). *Top 5 best python gui libraries*. Retrieved November 2, 2021, from https://www.askpython.com/python-modules/top-best-python-gui-libraries. (Cit. on pp. 36, 37)

Bais, R., Odek, P., & Ou, S. (2017). Sentiment classification on steam reviews. (Cit. on pp. 1, 13, 20, 24, 27).

Bangor, A., Staff, T., Kortum, P., Miller, J., & Staff, T. (2009). Determining what individual SUS scores mean: adding an adjective rating scale. *Journal of usability studies*, *4*(3), 114–123 (cit. on pp. 101, 105, 106).

Barbosa, J. L., Moura, R. S., & Santos, R. L. d. S. (2016). Predicting portuguese steam review helpfulness using artificial neural networks. *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*, 287–293 (cit. on pp. 1, 11, 20, 21, 27).

BBC. (2020). *Coronavirus: The world in lockdown in maps and charts*. Retrieved February 8, 2021, from https://www.bbc.com/news/world-52103747. (Cit. on p. 96)

Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with python: Analyzing text with the natural language toolkit*. " O'Reilly Media, Inc." (Cit. on p. 34).

Brooke, J. et al. (1996). Sus-a quick and dirty usability scale. *Usability evaluation in industry*, *189*(194), 4–7 (cit. on p. 122).

Brooke, J. (2020). SUS: A 'Quick and Dirty' Usability Scale. *Usability Evaluation In Industry*, (November 1995), 207–212. https://doi.org/10.1201/9781498710411-35 (cit. on p. 101)

Cai, J., Kapila, R., & Pal, G. (2000). Hmvc: The layered pattern for developing strong client tiers. *Java World*, *7*. https://www.infoworld.com/article/2076128/hmvc--the-layered-pattern-for-developing-strong-client-tiers.html (cit. on pp. 49, 50)

Cass, S. (2021). *Top programming languages 2021. python dominates as the de facto platform for new technologies*. Retrieved November 2, 2021, from https://spectrum.ieee.org/top-programming-languages-2021. (Cit. on p. 34)

Clement, J. (2018). *Distribution of steam users worldwide as of april 2018, by country*. https://www.statista.com/statistics/826870/steam-distribution-country/. (Cit. on p. 17)

Clement, J. (2021). *Number of games released on steam worldwide from 2004 to 2021*. https://www.statista.com/statistics/552623/number-games-released-steam/. (Cit. on p. 10)

D. Costa, C. (2020a). *Top 10 python gui frameworks for developers*. Retrieved November 2, 2021, from https://towardsdatascience.com/top-10-python-gui-frameworks-for-developers-adca32fbe6fc. (Cit. on pp. 36–38)

D. Costa, C. (2020b). *Top programming languages for data science in 2020*. Retrieved November 2, 2021, from https://towardsdatascience.com/top-programming-languages-for-data-science-in-2020-3425d756e2a7. (Cit. on p. 34)

Eberhard, L., Kasper, P., Koncar, P., & Gütl, C. (2018). Investigating helpfulness of video game reviews on the steam platform. *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 43–50 (cit. on pp. 1, 11, 12, 20, 27, 34).

Edward, S. (2021). *Sqlite vs mysql – what's the difference*. Retrieved November 9, 2021, from https://www.hostinger.com/tutorials/sqlite-vs-mysql-whats-the-difference/. (Cit. on p. 36)

Epp, R., Lin, D., & Bezemer, C. P. (2021). An Empirical Study of Trends of Popular Virtual Reality Games and Their Complaints. *IEEE Transactions on Games*, *13*(3), 275–286. https://doi.org/10.1109/TG.2021.3057288 (cit. on pp. 18, 20, 23, 27)

Erickson, A., Kim, K., Bruder, G., & Welch, G. F. (2020). Effects of Dark Mode Graphics on Visual Acuity and Fatigue with Virtual Reality Head-Mounted Displays. *Proceedings - 2020 IEEE Conference on Virtual*

Reality and 3D User Interfaces, VR 2020, 434–442. https://doi.org/10.1109/VR46266.2020.1580695145399 (cit. on p. 56)

Fagernäs, S., Hamilton, W., Espinoza, N., Miloff, A., Carlbring, P., & Lindner, P. (2021). What do users think about Virtual Reality relaxation applications? A mixed methods study of online user reviews using natural language processing. *Internet Interventions*, *24*(January). https://doi.org/10.1016/j.invent.2021.100370 (cit. on pp. 18, 27)

Fatima, H. (2017). *The 6 best python gui frameworks for developers*. Retrieved November 2, 2021, from https://blog.resellerclub.com/the-6-best-python-gui-frameworks-for-developers/. (Cit. on pp. 36, 37)

Fitzpatrick, M. (2020). Create GUI Applications whit Python & Qt5 (cit. on pp. 37–39, 49, 52, 56, 81, 90, 91).

Foxman, M., Leith, A. P., Beyea, D., Klebig, B., Chen, V. H. H., & Ratan, R. (2020). Virtual reality genres: Comparing preferences in immersive experiences and games. *CHI PLAY 2020 - Extended Abstracts of the 2020 Annual Symposium on Computer-Human Interaction in Play*, 237–241. https://doi.org/10.1145/3383668.3419881 (cit. on pp. 18, 27)

Gallinelli, N. (2021). *The 10 best data science programming languages to learn in 2021*. Retrieved November 2, 2021, from https://flatironschool.com/blog/data-science-programming-languages. (Cit. on p. 34)

Galyonkin, S. (2021a). *Api for steam spy*. Retrieved October 26, 2021, from https://steamspy.com/api.php. (Cit. on p. 22)

Galyonkin, S. (2021b). *How it works*. Retrieved October 26, 2021, from https://www.patreon.com/steamspy. (Cit. on p. 22)

GfK. (2020). Weekly unit sales growth of gaming virtual reality (vr) market in the united kingdom as impacted by covid-19 in weeks 8 to 12 of 2020 [graph]. Retrieved October 18, 2021, from https://www.statista.com/statistics/1123621/uk-gaming-vr-market-sales-growth-2020/. (Cit. on p. 18)

Group, G., & Museum, S. I. M. (2019). Video game metadata schema. (Cit. on p. 14).

Hamilton, D., McKechnie, J., Edgerton, E., & Wilson, C. (2021). *Immersive virtual reality as a pedagogical tool in education: a systematic literature review of quantitative learning outcomes and experimental design* (Vol. 8). Springer Berlin Heidelberg. https://doi.org/10.1007/s40692-020-00169-2. (Cit. on p. 19)

Hipp, R., Kennedy, D., & Mistachkin, J. (2021). *Sqlite fts5 extension*. Retrieved November 29, 2021, from https://www2.sqlite.org/fts5.html. (Cit. on pp. 61, 62)

Ho, J. C., & Zhang, X. (2020). Strategies for marketing really new products to the mass market: A text mining-based case study of virtual reality games. *Journal of Open Innovation: Technology, Market, and Complexity*, *6*(1), 1–14. https://doi.org/10.3390/joitmc6010001 (cit. on p. 18)

Ji, F. (2019). Sentiment analysis and opinion extraction of game reviews on steam (cit. on pp. 1, 12, 13, 20, 24, 27, 34).

Juniper Research. (2021). *Global video game market value from 2020 to 2025 (in billion u.s. dollars) [graph]*. Retrieved January 17, 2022, from https://www.statista.com/statistics/292056/video-game-market-value-worldwide/. (Cit. on p. 1)

Kamal, A. S. B., Saaidin, S., & Kassim, M. (2020). Recommender System: Rating predictions of Steam Games Based on Genre and Topic Modelling. *2020 IEEE International Conference on Automatic Control and Intelligent Systems, I2CACIS 2020 - Proceedings*, (June), 212–218. https://doi.org/10.1109/I2CACIS49202.2020.9140194 (cit. on pp. 1, 16, 20, 27, 34)

Kang, H.-N., Yong, H.-R., & Hwang, H.-S. (2017). A study of analyzing on online game reviews using a data mining approach: Steam community data. *International Journal of Innovation, Management and Technology*, *8*(2), 90 (cit. on pp. 1, 12, 20, 23, 24, 27).

Kasper, P., Koncar, P., Santos, T., & Gütl, C. (2019). On the role of score, genre and text in helpfulness of video game reviews on metacritic. *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 75–82 (cit. on pp. 7, 8, 27, 34).

Kaur, R. (2017). *Tableau competitors: Competitive analysis of top alternatives*. Retrieved November 23, 2021, from https://www.selecthub.com/business-intelligence/tableau-competitors/. (Cit. on p. 25)

Kay, R. H., & Loverock, S. (2008). Assessing emotions related to learning new software: The computer emotion scale. *Computers in Human Behavior*, *24*(4), 1605–1623. https://doi.org/10.1016/j.chb.2007.06.002 (cit. on pp. 100, 122)

Kivy Organization. (2021). *Kivy*. Retrieved November 7, 2021, from https://kivy.org/. (Cit. on p. 38)

Krasner, G. (1988). A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, *1*(3), 26–49 (cit. on p. 49).

Lam, K. (2020). *Full text search with sqlite*. Retrieved November 29, 2021, from https://kimsereylam.com/sqlite/2020/03/06/full-text-search-with-sqlite.html. (Cit. on p. 61)

LaViola Jr, J. J. (2000). A discussion of cybersickness in virtual environments. *ACM Sigchi Bulletin*, *32*(1), 47–56 (cit. on p. 19).

Leclair, D. (2021). *3 video game review sites you must look at before your next purchase*. Retrieved October 18, 2021, from https://www.makeuseof.com/tag/video-game-review-aggregators-purchase/. (Cit. on p. 5)

Lewis, R. J. (2000). An introduction to classification and regression tree (cart) analysis. *Annual meeting of the society for academic emergency medicine in San Francisco, California*, *14* (cit. on p. 12).

Li, X., Zhang, Z., & Stefanidis, K. (2021). A data-driven approach for video game playability analysis based on players' reviews. information 2021, 12, 129. (Cit. on pp. 17, 27).

Lin, D., Bezemer, C. P., Zou, Y., & Hassan, A. E. (2019). An empirical study of game reviews on the Steam platform. *Empirical Software Engineering*, *24*(1), 170–207. https://doi.org/10.1007/s10664-018-9627-4 (cit. on pp. 20, 23)

Lin, D., Bezemer, C.-P., & Hassan, A. E. (2018). An empirical study of early access games on the steam platform. *Empirical Software Engineering*, *23*(2), 771–799 (cit. on pp. 14, 20, 22, 23, 27).

Lombarti, A. (2022). *Tableau vs. powerbi: Which should you choose?* Retrieved January 17, 2022, from https://towardsdatascience.com/tableau-vs-powerbi-which-should-you-choose-1ad8d4d4be40. (Cit. on p. 25)

Martin, M. (2021). *Functional requirements vs non functional requirements: Differences*. Retrieved October 29, 2021, from https://www.guru99.com/functional-vs-non-functional-requirements.html. (Cit. on p. 29)

Muthyala, S. (2021). *10 best data science programming languages for data aspirants in 2021*. Retrieved November 2, 2021, from https://www.analyticsinsight.net/10-best-data-science-programming-languages-for-data-aspirants-in-2021/. (Cit. on p. 34)

Nederkoorn, C. (2021). *Top 10 python gui frameworks compared*. Retrieved November 2, 2021, from https://www.activestate.com/blog/top-10-python-gui-frameworks-compared/. (Cit. on pp. 36–38)

Nunes, M. (2020). *Tmodel-view-controller*. Retrieved November 15, 2021, from https://medium.com/@mestre.lider/model-view-controller-41a35614afe6. (Cit. on p. 49)

O'Neill, M., Vaziripour, E., Wu, J., & Zappala, D. (2016a). Condensing steam: Distilling the diversity of gamer behavior. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC, 14-16-Nove*, 81–95. https://doi.org/10.1145/2987443.2987489 (cit. on pp. 17, 23)

O'Neill, M., Vaziripour, E., Wu, J., & Zappala, D. (2016b). *Table and attribute descriptions*. Retrieved October 26, 2021, from https://steam.internet.byu.edu/. (Cit. on p. 23)

Paavilainen, J. (2020). Defining playability of games: Functionality, usability, and gameplay. *Proceedings of the 23rd International Conference on Academic Mindtrek*, 55–64 (cit. on p. 17).

Park, H., & Byun, H. (2016). Correlation analysis: Game professional score and user score on steam. *International Journal of Multimedia and Ubiquitous Engineering*, *11*(12), 237–246 (cit. on pp. 9, 27).

Pawlowski, L. (2019). *Easily embed secure power bi reports in your internal portals or websites*. Retrieved January 17, 2022, from https://powerbi.microsoft.com/nl-be/blog/easily-embed-secure-power-bi-reports-in-your-internal-portals-or-websites/. (Cit. on p. 25)

Pontiki, M., Galanis, D., Papageorgiou, H., Androutsopoulos, I., Manandhar, S., Al-Smadi, M., Al-Ayyoub, M., Zhao, Y., Qin, B., De Clercq, O., et al. (2016). Semeval-2016 task 5: Aspect based sentiment analysis. *International workshop on semantic evaluation*, 19–30 (cit. on p. 8).

PwC. (2020). Virtual reality (vr) gaming revenue worldwide from 2017 to 2024 (in billion u.s. dollars) [graph]. Retrieved October 18, 2021, from https://www.statista.com/statistics/499714/global-virtual-reality-gaming-sales-revenue/. (Cit. on p. 18)

PYPL. (2021). *Die beliebtesten programmiersprachen weltweit laut pypl-index im oktober 2021*. Retrieved November 2, 2021, from https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/. (Cit. on p. 33)

Python Software Foundation. (2021). *Python/c api reference manual*. Retrieved November 2, 2021, from https://docs.python.org/3/c-api/index.html. (Cit. on p. 35)

Radianti, J., Majchrzak, T. A., Fromm, J., Stieglitz, S., & vom Brocke, J. (2021). Virtual reality applications for higher educations: A market analysis.

Proceedings of the Annual Hawaii International Conference on System Sciences, *2020-January*, 124–133. https://doi.org/10.24251/hicss.2021.014 (cit. on pp. 18–21)

Radianti, J., Majchrzak, T. A., Fromm, J., & Wohlgenannt, I. (2020). A systematic review of immersive virtual reality applications for higher education: Design elements, lessons learned, and research agenda. *Computers and Education*, *147*(November 2019), 103778. https://doi.org/10.1016/j.compedu.2019.103778 (cit. on pp. 19, 27)

Sánchez, J. L. G., Vela, F. L. G., Simarro, F. M., & Padilla-Zea, N. (2012). Playability: Analysing user experience in video games. *Behaviour & Information Technology*, *31*(10), 1033–1054 (cit. on p. 17).

Scheiner, M. (2021). *Top 15 best data analytics tools & software comparison 2022*. Retrieved January 17, 2022, from https://crm.org/news/best-data-analytics-tools. (Cit. on p. 25)

Scott, T. (2017). *16 tableau alternatives for visualizing and analyzing data*. Retrieved November 23, 2021, from https://technologyadvice.com/blog/information-technology/tableau-alternatives/. (Cit. on p. 25)

Sherrick, B., & Schmierbach, M. (2016). The effects of evaluative reviews on market success in the video game industry. *The Computer Games Journal*, *5*(3), 185–194 (cit. on pp. 9, 27).

Sifa, R., Bauckhage, C., & Drachen, A. (2014). Archetypal game recommender systems. *LWA*, 45–56 (cit. on p. 20).

Sifa, R., Drachen, A., & Bauckhage, C. (2015). Large-scale cross-game player behavior analysis on steam. *Proceedings of the 11th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2015*, *2015-Novem*, 198–204 (cit. on p. 20).

Smutny, P., Babiuch, M., & Foltynek, P. (2019). A review of the virtual reality applications in education and training. *Proceedings of the 2019 20th International Carpathian Control Conference, ICCC 2019*, 2019–2022. https://doi.org/10.1109/CarpathianCC.2019.8765930 (cit. on pp. 18, 19, 27)

Sobkowicza, A., & Stokowiec, W. (2016). Steam Review Dataset - new, large scale sentiment dataset. *Emotion and Sentiment Analysis PROCEEDINGS*, *95*, 55–58. http://gsi.dit.upm.es/esa2016/Proceedings-ESA2016.pdf (cit. on pp. 20, 21, 24)

Stack overflow. (2012). *Can i read and write to a sqlite database concurrently from multiple connections?* Retrieved November 16, 2021, from https:

//stackoverflow.com/questions/10325683/can-i-read-and-write-to-a-sqlite-database-concurrently-from-multiple-connections. (Cit. on p. 52)

Steam. (2021). *Number of peak concurrent steam users from january 2013 to september 2021 (in millions) [graph]*. Retrieved January 17, 2022, from https://www.statista.com/statistics/308330/number-stream-users/. (Cit. on p. 1)

Steam Spy. (2021). *Number of games released on steam worldwide from 2004 to 2021 [graph]*. Retrieved January 17, 2022, from https://www.statista.com/statistics/552623/number-games-released-steam/. (Cit. on p. 1)

Stegner, B. (2021). *The 7 best gaming news sites and game review sites*. Retrieved October 18, 2021, from https://www.makeuseof.com/tag/10-websites-latest-game-reviews-gaming-news/. (Cit. on p. 5)

Stinner, V. (2017). *Build cpython on windows*. Retrieved November 29, 2021, from https://cpython-core-tutorial.readthedocs.io/en/latest/build_cpython_windows.html. (Cit. on p. 60)

Strååt, B., Verhagen, H., & Warpefelt, H. (2017). Probing user opinions in an indirect way: An aspect based sentiment analysis of game reviews. *Proceedings of the 21st International Academic Mindtrek Conference*, 1–7 (cit. on pp. 7, 8, 27).

Toy, E. J., Kummaragunta, J. V., & Yoo, J. S. (2018). Large-scale cross-country analysis of steam popularity. *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 1054–1058 (cit. on pp. 17, 20, 24, 26, 27).

Valve. (2021a). *Authentication using web api keys*. Retrieved October 25, 2021, from https://partner.steamgames.com/doc/webapi_overview/auth. (Cit. on p. 22)

Valve. (2021b). *Error codes & responses*. Retrieved October 26, 2021, from https://partner.steamgames.com/doc/webapi_overview/responses. (Cit. on p. 22)

Valve. (2021c). *Isteamapps interface*. Retrieved October 25, 2021, from https://partner.steamgames.com/doc/webapi/ISteamApps. (Cit. on p. 22)

Valve. (2021d). *User reviews - get list*. Retrieved October 25, 2021, from https://partner.steamgames.com/doc/store/getreviews. (Cit. on p. 22)

Valve. (2021e). *User reviews - get list*. Retrieved December 20, 2021, from https://partner.steamgames.com/doc/store/getreviews. (Cit. on p. 72)

Wang, D., Moh, M., & Moh, T.-S. (2020). Using deep learning and steam user data for better video game recommendations. *Proceedings of the 2020 ACM Southeast Conference*, 154–159 (cit. on pp. 1, 15, 16, 20, 23, 27).

Willison, S. (2021). *Tips for getting sqlite under python to perform better with more ram?* Retrieved November 16, 2021, from https://sqlite.org/forum/info/f6bdd6b503cd4347. (Cit. on p. 52)

Windleharth, T. W., Jett, J., Schmalz, M., & Lee, J. H. (2016). Full steam ahead: A conceptual analysis of user-supplied tags on steam. *Cataloging & Classification Quarterly*, *54*(7), 418–441 (cit. on pp. 13, 20, 27).

wxPython Team. (2020). *Overview of wxpython*. Retrieved November 7, 2021, from https://wxpython.org/pages/overview/. (Cit. on p. 38)